Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Sergio Soto Torres

Front-End design with a database interface

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Sergio Soto Torres

Front-End design with a database interface

Bachelor Thesis based on the examination and study regulations for the Bachelor of Engineering degree programme Information Engineering at the Department of Information and Electrical Engineering of the Faculty of Engineering and Computer Science of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. Ing Franz Schubert
Second examiner : Prof. Dr. Martin Zapf

Date of delivery July 24th

Sergio Soto Torres

Title of the paper

Front-End-Design with a database interface

Keywords

PostgreSQL, Django, Python, Programming language, Nginx, Web-Interface, Smart Heat Grid Hamburg, JavaScript, jQuery, HTML, CSS, PyCharm, pgAdmin

Abstract

This Bachelor Thesis is about the creation of a complete web server that can be used by the team of the Smart Heat-Grid Hamburg to manage the different projects and libraries that they have, and specially to parameterize simulation models.

It is important to have a friendly and flexible web interface in order to make the correct simulations for developing the intelligent smart heat grid for this big project.

The server is created in a PC with Ubuntu located at the server room of the CC4E building, in Bergedorf, Hamburg. It must have a connection with the existing database of the CC4E where all the data will be stored.

It was needed to select a programming language and different tools that can be used in other projects of the centre because they will be related with my work.

Sergio Soto Torres

Thema der Bachelorthesis

Front-End-Design mit einer Datenbankschnittstelle

Stichworte

PostgreSQL, Django, Python, Programming language, Nginx, Web-Interface, Smart Heat Grid Hamburg, JavaScript, jQuery, HTML, CSS, PyCharm, pgAdmin

Kurzzusammenfassung

In dieser Bachelor-Thesis wird eine Web-Anwendung entwickelt, mit der das Team des Smart-Heat-Grid-Hamburg verschiedene Projekte und Bibliotheken verwalten und spätere Simulationsmodelle parametrisieren kann.

Wichtig ist, eine einfach zu bedienende aber dennoch flexible Web-Schnittstelle zu entwickeln, um fehlerfrei Simulationen für das intelligente, große Smart-Heat-Grid-Projekt erstellen zu können.

Der Web-Server ist mit einem PC und Ubuntu umgesetzt. Dieser befindet sich im Serverraum des CC4E Gebäudes in Hamburg Bergedorf. Der Web-Server muss eine Verbindung mit der bestehenden Datenbank des CC4E herstellen, da auf dieser alle simulationsrelevanten Daten gespeichert werden. Der Web-Server soll über das Internet, also außerhalb des lokalen Netzwerks, zur Verfügung stehen.

Für diese Thesis wurden die genutzte Programmiersprache und die unterschiedlichen Tools so ausgewählt, dass diese in anderen, mit meiner Arbeit verbunden, Projekten des Forschungszentrums weiter genutzt werden können.

Acknowledgement

First I would like to thank the support given by Philipp Eike Janßen and all the team of researchers of the CC4E for developing this project and solving the doubts about the process and their previous work in the database

Also I would like to thank the Prof. Dr.-Ing Franz Schubert and the Prof. Dr.-Ing Lutz Leutelt for giving me this chance to work in this important research centre with projects of different branches than mine, which made me learn a lot.

Finally, I would like to thank the entire C4DSI team for all the help and to all the support received from the HAW for staying in Hamburg and developing this thesis.

Table of Contents

Figures

1 Introduction

The aim of this thesis is to start the design and development of a frontend web service to parametrize simulation models.

1.1 Motivation

There are many reasons to choose a project like this to do my Bachelor Thesis and to finish my major here in Hamburg.

As Germany is a leading country in engineering, it will be useful for my future to have an experience abroad like this.

Also, being part of a research centre like the CC4E and working for a real big project instead of doing other tasks where the solution is known before, makes it more challenging but more exciting aswell.

Another good point about researching here is that I can learn about other branches of engineering as this centre is focused on renewal energies and this thesis is about computer science.

I had to work with programming languages and tools that were new for me, so it has been hard to learn about them in a short time but also very useful to increase my knowledge in this important field for engineers.

1.2 Approach

From the beginning, I set three main goals in order to work more efficiently and to define the structure of my thesis.

- ➔ **Goal 1**. Set up a web server with Nginx and Ubuntu Server in one of the computers of the server room of the CC4E building. It is important to install Django there when the server is available from outside of the local network.
- ➔ **Goal 2**. Create a Python web interface with the Django framework that has connection with the PostgreSQL database of the CC4E research centre and that can be deployed in the Nginx server. It is a complicated part because I have to connect all these tools that must work together.
- ➔ **Goal 3**. Develope different functionalities in the web server that can be useful for the management of different projects and libraries and specially the simulation of parameters.

2 Preparatory work

First of all, it was needed to learn about the different tools that I was going to work with and to decide if the following tasks could be done with them.

As the Energy Campus of the CC4E is a very new building, their projects are also starting, so at the beginning it was needed to choose with which programming language I had to work with and which other tools I could use. It was an important point because future tasks of this big project will depend on my work.

The chosen programming language (Python [1] with the framework Django [2]) and the Nginx [3] software were new to me, so I had to learn to learn and research about them before starting my work here.

2.1 Nginx

Figure 2.1: Nginx logo

Nginx [3] is a open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. I used it as the web server for this project.

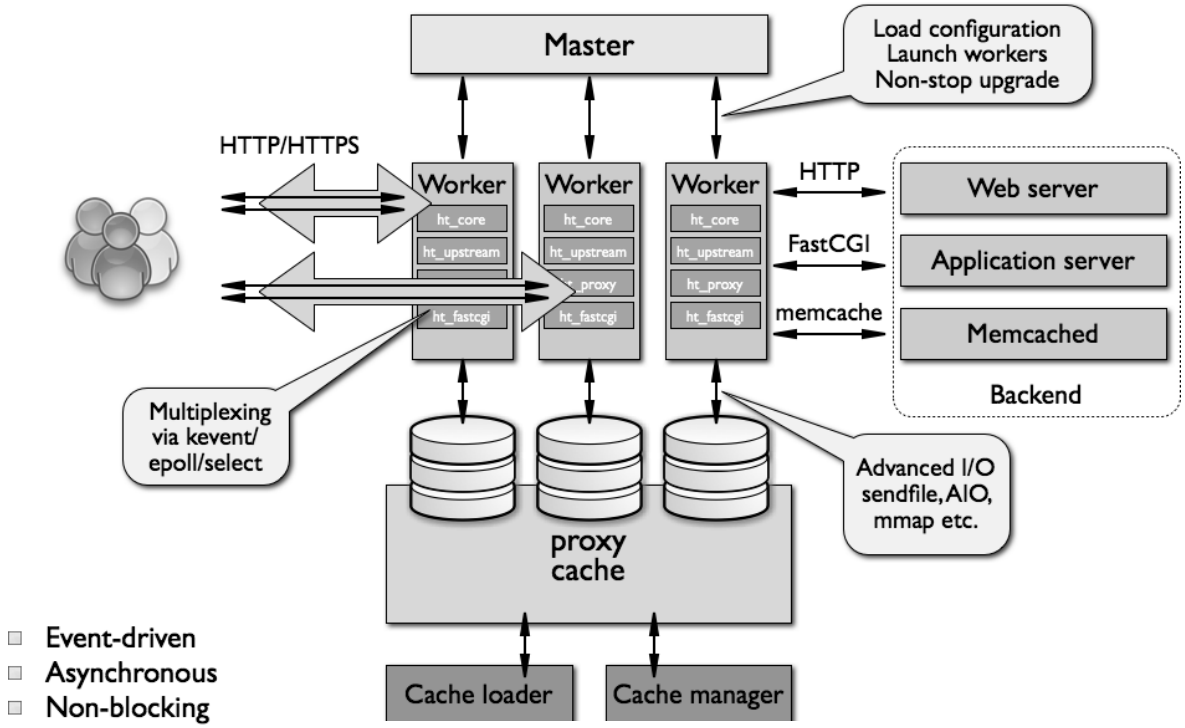

Figure 2.2: Nginx architecture [31]

According to Netcraft, Nginx served or proxied 29% of the busiest sites in May 2017, including well known sites as Netflix, wordpress.com and Facebook.

Main Characteristics

- ➔ Efficience: Light web server that does not consume a lot of resources.
- ➔ High performance: it can serve high number of request per second.
- ➔ Flexibility: it has many options to set.
- ➔ Stability: Nginx can bear large amounts of traffic

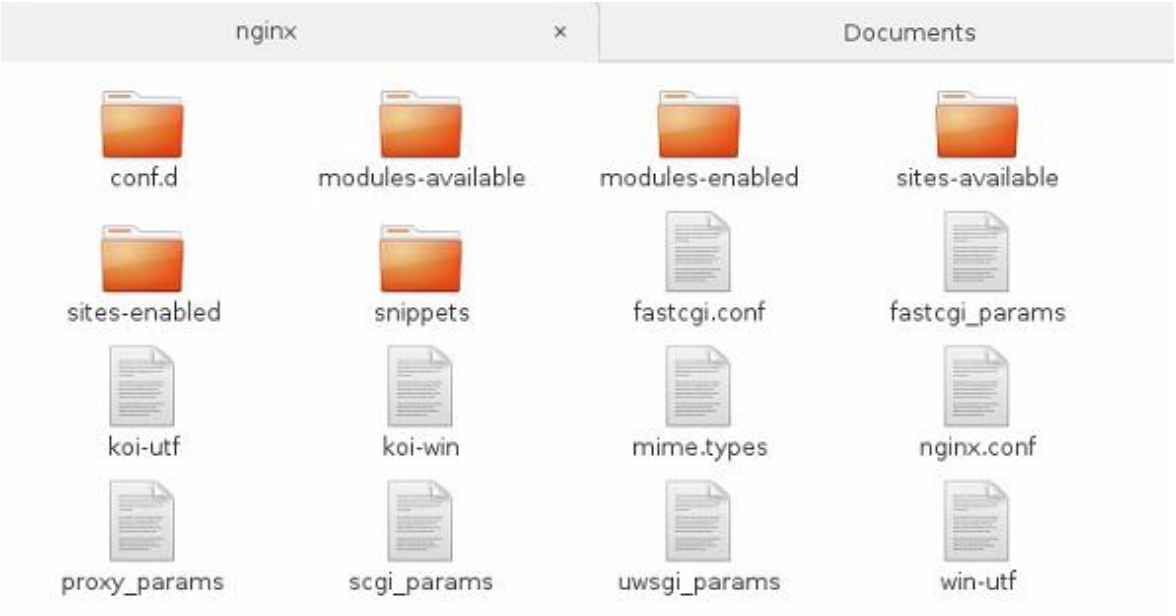

Figure 2.3: Default folders in Nginx

2.2 PuTTy

Figure 2.4: PuTTy logo

PuTTy [4] is a SSH [5] client that acts as a terminal simulator. At first it was available only for Windows but nowadays it is also available for some Unix platforms.

It was written and maintained by Simon Tatham.

Main Characteristics

- ➔ Hosts and preferences storage for further uses.
- ➔ PSCP and PSFTP: line command clients for SCP and SFTP
- ➔ Control of the port redirection with SSH (including X11).

2.3 X11

The X Windows System [6] is a windowing system for bitmaps displays. It provides the basic framework for a GUI environment such as drawing and moving windows on the display device and interacting with a mouse and keyboard. In this project I used Xming.

2.3.1 Xming

Xming [7]: it is a X11 display server for Microsoft Windows operating systems. Xming may be used with implementations of SSH to secure forward X11 sessions from other computers. It supports PuTTy.

2.4 Python

Figure 2.5: Python logo

Python [1] is a high-level programming language which is widely used for general-purpose programming as it supports different types of programation (object oriented, imperative and functional). Thus, with it we can create applications, network servers or web pages.

It was created by Guido Van Rossum and first released at 1991. The chosen name, Python, is a reference to the British comics Monty Python. Nowadays it is one of the most popular programming languages.

Main Characteristics

- ➔ Many libraries containing a wide variety of data and functions. It makes easier to develop some tasks as it is not needed to program them from zero.
- ➔ Simplicity: a program in Python uses to have less code than an equivalent program in Java or C, so it is more simple, clean and fast.
- ➔ Many platforms: we can develop in Unix, Windows, Mac, etc.
- ➔ FLOSS: Free and Open Source Software
- ➔ Integration: it can be included in other programs with different languages.

2.5 Django

Figure 2.6: Django logo

Django [2] is a Python Web framework that includes many interesting tools for a fast and clean development of web applications. The main goal of Django is to make easier the creation of complex websites with database.

It was created in 2003 by Adrian Holovaty and Simon Willison and its name is a reference to the jazz musician Django Reinjardt.

It is based in the Model-View-Controller architecture (MVC), which is a software architectural pattern very popular for designing web applications.

It basically consists of these three parts:

- ➔ Model: Object-relational mapper (ORM) that mediates between data models and a relational database
- ➔ View: System for processing web requests with different views (templates).
- ➔ Controller: Regular-expression-based URL dispatcher

Main Characteristics

- ➔ Fast learning curve: it is easy and fast for the developers to take applications from concept to completion.
- ➔ Many predefined tools that can be used to handle common Web development tasks such as user authentication, content administration or site maps.
- ➔ Admin interface: Django includes a complete interface for. Manage many settings in the web server. It includes a direct connection with the database.
- ➔ DRY: Don't Repeat Yourself
- ➔ URLs dispatcher

2.5.1 Architecture

These are the different files and directories that we obtain after installing Django and starting a new site:

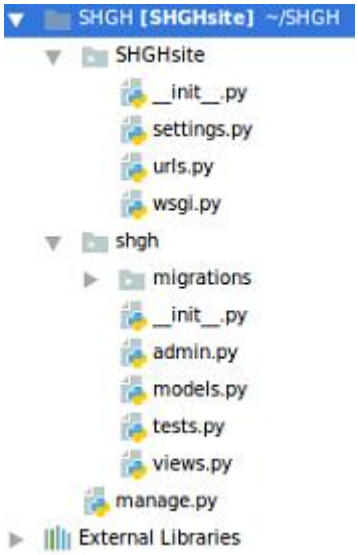

Figure 2.7: Default Django files and directories

Settings.py	Connection with the database and other configuration options that we can set,k but apparently with some restrictions.
Urls.py	„Url“ adresses that we can call through the HTML templates (they should be created by us in another directory) in order to open a function in the controller („views.py“ is the default one).
Admin.py	Automatic admin interface that reads metadata from the models of the project. Trusted users can easily manage the content on the site through this interface.
Models.py	Source of information about the data. It contains the essential fields and behaviors of the data that we want to store. Each model is a Python class and each attribute represents a database field.
Views.py	Controller of our project, where we create the different Python functions in order to accept inputs (web requsts) and to convert them to comands (web responses) for the model or the views (HTML templates)
Manage.py	Tool that can be used for interacting with a project in Django. Different command lines can be called, such as: · „help“: request for help · „version

Table 1: Explanation of the default Django files

After this, the different HTML templates can be created in another directory, and other python files can be created aswell.

The connection between the most important files and the database in Django can be briefly described with the following graphic:

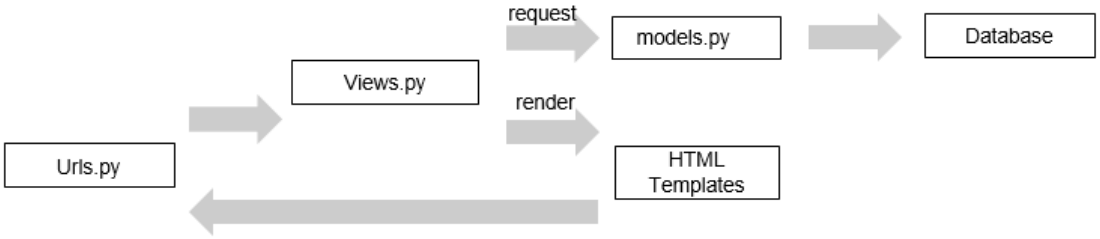

Figure 2.8: Graphic of the basic Django architecture

2.6 PyCharm

Figure 2.9: PyCharm logo

Pycharm [8] is an IDE developed by the company JetBrains.

It is a very complete program that has many tools for working with Python projects.

Main Characteristics

- ➔ Integration with Python frameworks such as Django or Flask.
- ➔ Integration with JavaScript.
- ➔ Autocomplete, syntax highlighting and analysis and refactoring tools.

2.7 HTML5

Figure 2.10: HTML5 logo

HTML5 [9] is the last version of HTML (Hyper Text Markup Language), a very common language for the creation of websites as it is one of the three core technologies of World Wide Web production, alongside JavaScript snd CSS.

In this version there are new elements like the canvas [10], which is very important for this project.

2.8 JavaScript

Figure 2.11: JavaScript logo

JavaScript (JS) [11] is a high-level interpreted language. This kind of languages are ideal for web production because the web browsers are the ones that interpretate and execute these programs.

2.8.1 JQuery

jQuery [12] is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animation, handle events and develop Ajax [13] applicatons.

2.9 CSS

Figure 2.12: CSS logo

CSS (Cascading Style Sheets) [14] is a language used to define the presentation design of an application written in HTML (or also XML [15] and XHTML [16]). It is important in order to separate the design and the structure of an application.

It can be applied in three different ways:

- ➔ CSS in-line: the design is embedded in the application code, as a property.
- ➔ CSS intern: the design is embedded in the application but with its own place.
- ➔ CSS extern: the design is in an extern file, completely separated.

2.10 PostgreSQL

Figure 2.13: PostgreSQL logo

PostgreSQL [17] is a open source object-relational database system.

It uses a client/server model and multiprocessing for giving stability to the system. It means that if there is some error in one process, it will not affect to the rest of the system.

Main Characteristics

- ➔ Stability: it can bear years of high activity operation without crashing.
- ➔ Cross platform: available for almost every brand of Unix and it has Windows compatibility.
- ➔ Designed for high volume environments: it is extremely responsive in high volume environments.
- ➔ GUI database design and administration tools: there are many GUI Tools available for PostgreSQL. In this project I used pgAdmin.

2.11 PgAdmin

PgAdmin [18] is an application that allows us to manage a PostgreSQL database with a graphical interface.

2.11.1 SHGH Database

The CC4E has a PostgreSQL server with users with different roles. There are employees of the Energy Campus, students of the HAW and employees of other companies such as Energy Hamburg.

As my thesis topic is part of the Smart Heat-Grid Hamburg project, I have to work with the SHGH database and its tables. These are the predefined tables involved in my task that they had:

- ➔ Public:
  This is the default schema where the Django tables are created after installing it.
- ➔ Smardism_project_db_dev_pj:
  In this schema we have the columns concerning the projects: Folders, projects, variants and scenarios.

Folders	**Projects**	**Variants**
Folder_name (Primary Key)	Project_name (PK)	Variant_name (PK)
date	Folder_name (Foreign Key)	Folder_name (FK)
Folder_owner	date	Project_name (FK)
	Project_owner	Date
		Variant_owner

Table 2: Tables of the „smardism_project_db_dev_pj“

- ➔ Smardism_libraries_db_dev:
  Here there are the types that will be used in the PI Editor and also the quantities and methods.

Types	**Quantities**	**Methods**
Type_name (PK)	Quantity_name (PK)	Method_name (PK)
Type_date	Unit	Type_name (FK)
Image_url	Datetype	Type_date (FK)
	Color	Method_code
		Method_date

Table 3: Tables of the „smardism_libraries_db_dev“

➔ Smardism_pi_db_dev:
In this schema we will store the different components that we create in the PI Editor.

Components		
Component_name (PK)	Type_name (FK)	height
Component_date	Type_date (FK)	width
Folder_name (FK)	Pos_x	box
Project_name (FK)	Pos_y	

Table 4: Table of the „smardism_libraries_db_dev“

The Primary Key [19] is a field in one table that uniquely identifies each row in a table. There can not be two rows with the same primary key.

The Foreign Key [20] is a field in one table that uniquely identifies a row of another table or the same table.

3 Set up of the server

A web server faces the outside world. It can serve files directly from the file system but it can not talk directly to applications made with Django.

For setting up of the web server I needed to use Nginx along with a Web Server Gateway Interface (WSGI) [21], which is a Python standard. It was required in order to set it up as a production server. Without this process, this server could be only reachable inside of its local network.

In this case I set up Uwsgi [22], aWSGI implementation, so that it creates a Unix socket and serves responses to the web server via the uWSGI protocol:

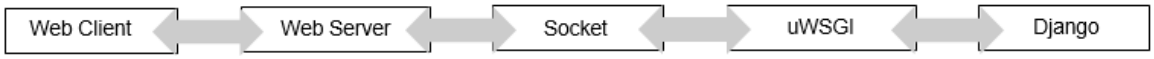

Figure3.1: Schema of the connections of the web server.

I also needed to install a virtual environment [23] before the process.

3.1 Set Up of the working environment

First of all, I had to install Ubuntu Server [24] in one of the free computers in the Server room. But for working with it I needed another computer in another room, in this case a small one with Windows and sharing the same network.

The connection between both was made with the software PuTTy [4]. It is a SSH [5] client that acts as a terminal simulator, so I could control the server computer with command lines just after log in with the user predefined there.

But for doing a project like this I wanted to have also a graphical interface and I installed Xming [7], a X11 [6] display server for Windows that let me open graphical windows of the Ubuntu computer through PuTTy.

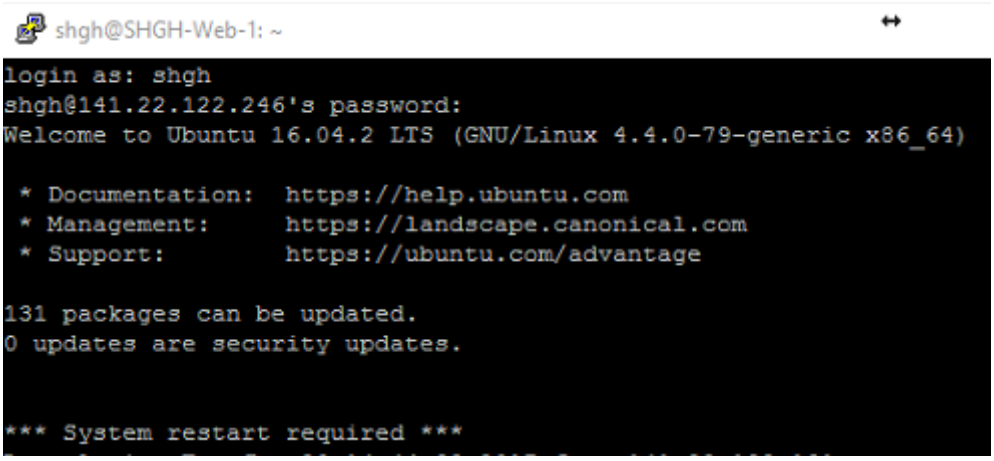

Figure 3.2: Connection with PuTTy

For opening a session in PuTTy, it is necessary to write the host name (or IP address) of the computer that we want to control.

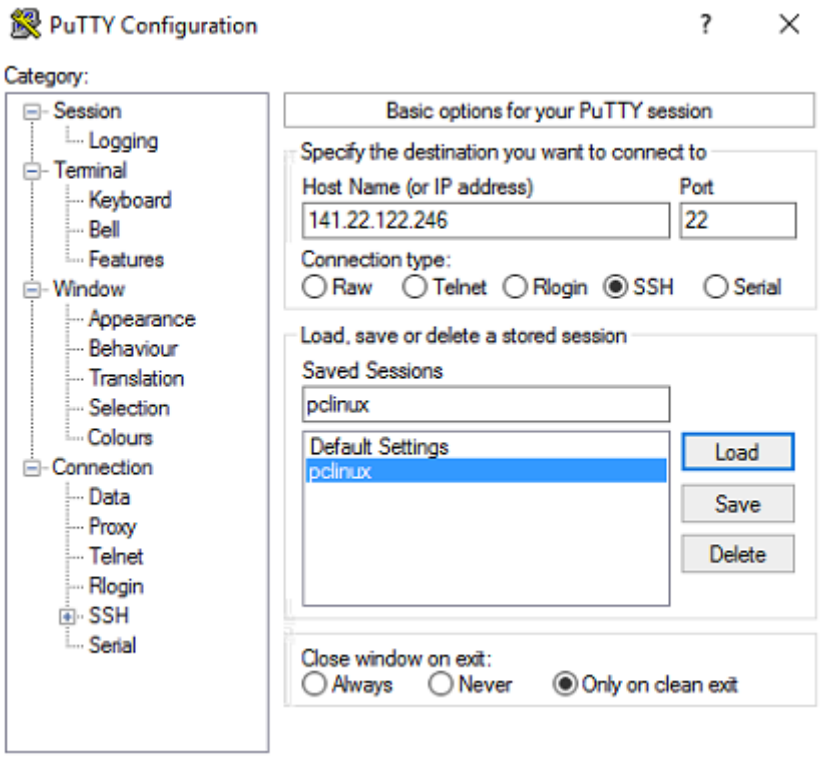

Figure 3.3: PuTTy configuration

But before opening I went to the SSH section for enable the X11 forwarding option in order to open graphical interfaces during my session in PuTTy just using Ubuntu commands.

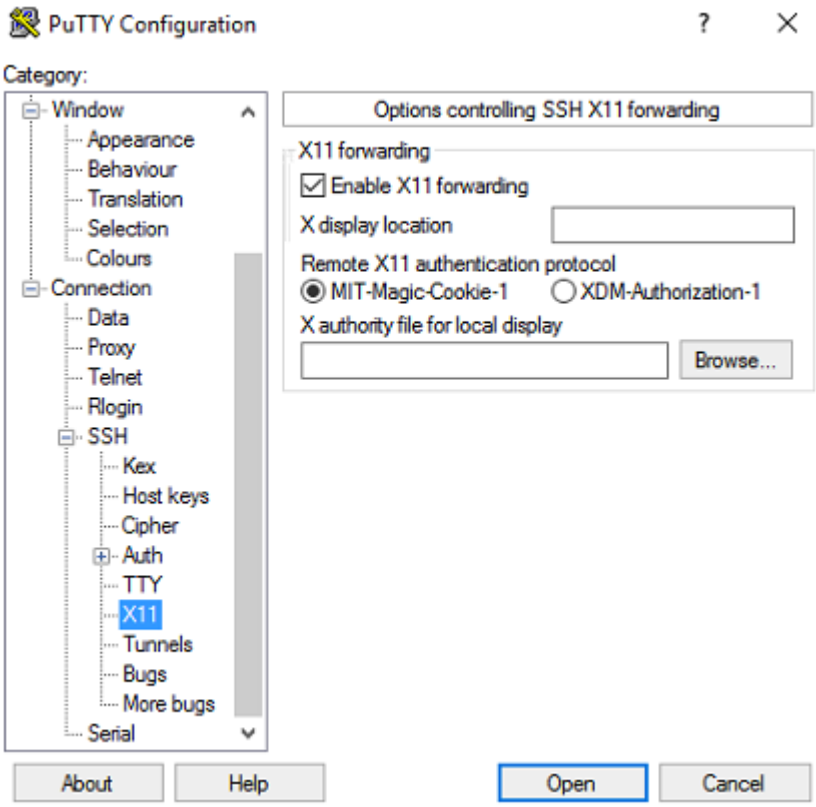

Figure 3.4: Connection with PuTTy (X11 forwarding)

After that we can open the session and write the correct username and password of the remote computer (it is also needed for „sudo“ [25] commands).

3.2 Set Up of Web Server

After that I was able to start the setting up of the web server integrating our different tools (Nginx, Django, uWSGI and the virtual environment).

I created a virtual environment to install all the software there:

```
sudo pip install virtualenv
virtualenv myvenv
cd myvenv
source bin/activate
```

Then I installed Django there and I created the site „SHGHamburg“ where I was going to store all of the project. Then I installed the uWSGI :

```
sudo pip install Django
django-admin.py startproject SHGHamburg
cd SHGHamburg
pip install uwsgi
```

I tested that the connection between Django and uWSGI was correct with the following command that loads the specific wsgi module in the localhost:

```
uwsgi --http :8000 --module mysite.wsgi
```

Then just pointing the browser at the IP 141.22.122.246:8000 I could check that the connection was working at that moment and I was able to install Nginx.

```
sudo apt-get install nginx
sudo /etc/init.d/nginx start
```

For configuring the site, first of all I downloaded and copied into my project directory the file „wsgi_params“, which is available in the nginx directory of the uWSGI distribution. Also in the project directory I created a file called „SHGHamburg_conf_nginx.conf“ where I use Unix sockets for doing the connection between Django and Nginx.

```
# SHGHamburg_conf_nginx.conf

# the upstream component nginx needs to connect to
upstream django {
    server unix:///home/shgh/SHGHamburg/SHGHamburg.sock; # for a file socket
}

# configuration of the server
server {
    # the port your site will be served on
    listen      80;
    # the domain name it will serve for
    server_name .http://141.22.122.246; # substitute your machine's IP address or FQDN
    charset     utf-8;

    # max upload size
    client_max_body_size 75M;   # adjust to taste

    # Django media
    location /media  {
        alias /home/shgh/SHGHamburg/media;  # your Django project's media files - amend as required
    }

    location /static {
        alias /home/shgh/SHGHamburg/shgh/static; # your Django project's static files - amend as required
    }

    # Finally, send all non-media requests to the Django server.
    location / {
        uwsgi_pass  django;
        include     /home/shgh/SHGHamburg/uwsgi_params; # the uwsgi_params file you installed
    }
}
```

This configuration file tells Nginx to serve up media and static files from the system and also handle requests where Django is involved. The media files are the images of the types that we have in the database and the static files are the stylesheets of the project (we only have one for now, „shgh.css“). These files are stored in their own directory whcih must be defined in the Django file „settings.py“ aswell.

In order that nginx can see the file, I made a symlink [26]from /etc/nginx/sites-enabled with the command „ln -s“

Before starting Nginx, it is needed to collect all Django static files in the static folder. It is done through the „settings.py“file and then running „python manage.py collectstatic”:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static/")
```

Next step is to restart nginx and run the application through this command that sets a very permissive configuration of the socket:

```
uwsgi --socket SHGHamburg.sock --module SHGHamburg.wsgi --chmod-socket=666
```

4 Creation of the Web Interface

In this part of the document I will explain which are the first steps of my task in the whole project and what is the point from where I have to start.

4.1 Connection with the database

The most important struggle that I had at the beginning of this project was to make the connection between the web server that I was starting to create and the database of the research centre.

Django provides a simple an easy connection between this framework and a new MySQL or PostgreSQL database, but it seems more difficult if the database is already created. Thus, I spent some time researching about how could I make the connection just after the login function, in order that every user of the CC4E database can access with their own data.

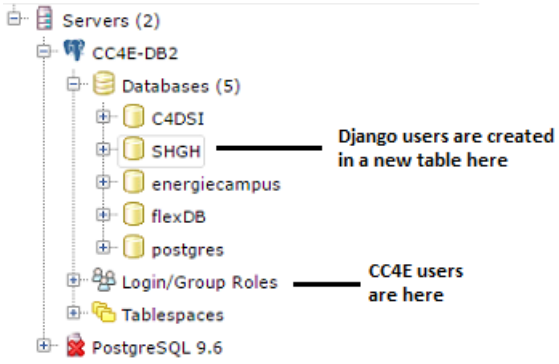

Figure 4.1: Location of the users of the CC4E database and the Django users

The problem was that Django is designed to create new users in new tables in the database and these users are the ones that can manage the admin interface.And for the database just one user is needed, as shown here:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'SHGH',
    'OPTIONS': {
                'options': '-c search_path=smardism_project_deb_dv'
            },
        'USER': 'shg.hamburg',
        'PASSWORD': 'shgh#',
        'HOST': '141.22.122.14',
        'PORT': '5432',
    }
}
```

Figure 4.2: Django method for connecting with the database in “settings.py”

Finally, with the help of the engineers working at the Campus for understanding the database that they previously created, the solution I found was to create a new controller file called „*db_pointer*“ [27] that works as a intermediary between the Django application and the PostgreSQL database. It is written in pure Python language and does not render any template.

In this file, I created a class called PiDBConnector for storing the different functions that are related with the database. So every time that it is needed to request or change (create, update, delete) some information from the database the PiDBConnector will be used. This Python module works in the project as the subsitute of the „models.py“ file that Django uses as an intermediary between the application and the database (a model in the MVC pattern).

I could not work with the „models.py“ file because the classes written there define new tables for the database in the schema selected in the „setting.py“ module. Thus, I did not know how to work with existing tables from different schemas instead of creating new ones as Django does.

At the beginning, a init function was needed:

```
1.  # dbPointer.py
2.  class PiDBConnector:
3.
4.
5.      def __init__(self,dbHostAddr, dbName, dbUser, dbPw, dbdataTimeZone, dbPrimKey
    s=dict(), dbTable=""):
6.          self.dbHostAddr = dbHostAddr
7.          self.dbName = dbName
8.          self.dbUser = dbUser
9.          self.dbPw = dbPw
10.         self.dataTimeZone = dbdataTimeZone
11.
12.         # Database specific Datastructures
13.         self.dbTable       = dbTable
14.         self.dbPrimKeys    = dbPrimKeys
15.         self.cur           = None
16.         self.conn          = None
```

Then I created another function for doing the connection to the database:

```
1.  # dbPointer.py
2.  def connect(self):
3.      try:
4.          self.conn = psycopg2.connect("dbname={0} hostaddr={1} user={2}
            password={3}".format(self.dbName, self.dbHostAddr,
            self.dbUser, self.dbPw))
5.              print("Connection to Database established.")
6.              print("connect to PostgrSQL-DB {0} at {1}, with User {2}"
                .format(self.dbName, self.dbHostAddr, self.dbUser))
7.              print(self.dbHostAddr, self.dbName, self.dbUser,
                self.dbPw, self.dataTimeZone)
8.              self.cur = self.conn.cursor()
9.          except:
```

```
            print("Could not connect to Database.")
            print(self.dbHostAddr, self.dbName, self.dbUser,
            self.dbPw, self.dataTimeZone)

        return (self.conn, self.cur)
```

The following step was to create a function in the controller („views.py“) that requests this information to the PiDBConnector („dbpointer.py“) and send it to a template. Global variables were needed as well for storing the user login data (username and password) during his session. They will be available for the other functions in the controller, as with this method the connection to the database has to be restarted every time that the we call a new function.

For calling a function in the dbPointer first we have to define a new variable „myDbConnector“ inside of the function in the controller and render to the init function the host address, the name of the database, the time zone and the information that the user sends (username and password). After that, we are able to call any function from the pointer. In this case we have to call the function „connect“ explained before.

```
# views.py
def login(request):

    form = User(request.POST or None)

    if form.is_valid():  # if the user is sending correct data
        data = form.cleaned_data
        name = data.get("username")
        password = data.get("password")

        # We store the username and password in global variables in order to use
        them in the other functions
        global name_conn
        global password_conn
        name_conn = name
        password_conn = password

        # Connection with the database
        myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name, dbPw=password, dbdataTimeZone="CET")

        try:
            # if the username and the password are correct we proceed to connect
            myDbConnector.connect()
            return render(request, 'shgh/select.html', {'username': name_conn})

        except:
            # if there is some error in the login
            error = True
            form = User()
            context = {'form': form,
                       'error': error}
            return render(request, 'shgh/index2.html', context)

    return render(request, 'shgh/index2.html')
```

In the function, the username and password are taken from the web interface where the user writes them in text inputs. Then, after setting the connector with the database, the system tries to connect to the database with the selected username and password

and the database host address (141.22.122.14), the name of the database where we are working (SHGH) and the data time zone (CET).

4.2 Templates

These are the main templates that the web interface is supposed to have:

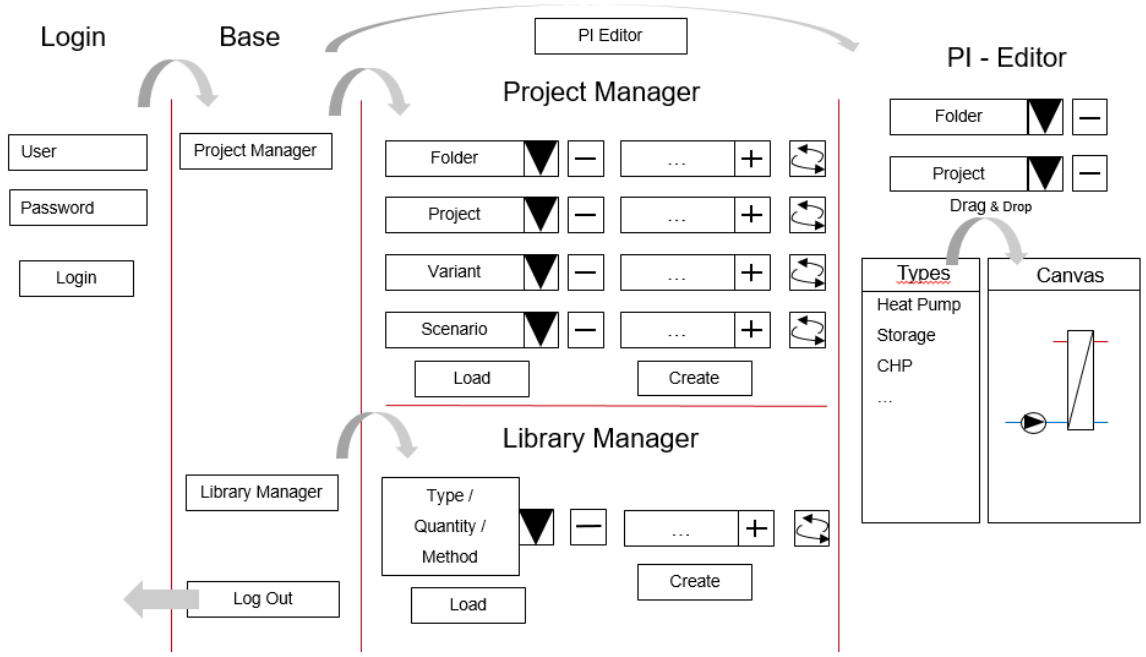

Figure 4.3: Schema showing the main templates of the web interface.

Index	First template that the server shows when the user access. It makes the connection with the database (through a function in the controller).
Home	„Url“ adresses that we can call through the HTML templates (they should be created by us in another directory) in order to open a function in the controller („views.py“ is the default one).
Project Manager	The user can create or load a project, defining in which folder will be stored, and also its variants and scenarios. When the user loads a project, the PI –Editor template will be opened.
Library Manager	The user can create or load a type, quantity, parameter and method.
PI - Editor	The user can create or load a component inside of the previously selected project.

Table 5: Explanation of the main templates of the web interface.

4.3 Base template

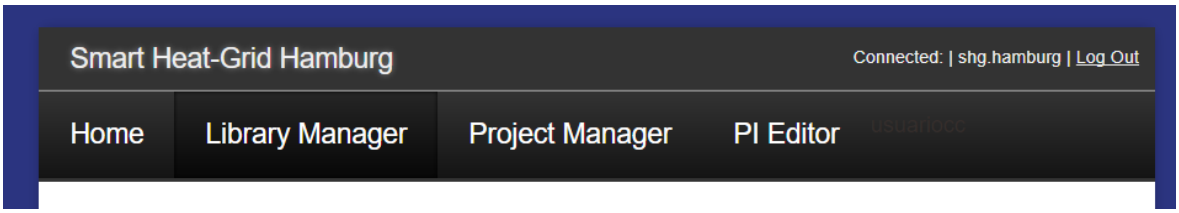

Figure 4.4: Menu of the Base template

The base template is one that will be extended to all the rest (except the index template). It includes a header and a body where there is a menu that let us easily access to the main templates. It also shows who is the user who is connected in this session and there is the option to log out.

The menu is created using CSS [14] for the design. Here is a little extract located in the „shgh.css“, the stylesheet where all the design of the application is defined.

```
# shgh.css
#cssmenu ul, #cssmenu li, #cssmenu span
{
  margin: 0;
  padding: 0;
  position: relative;
}
#cssmenu {
  line-height: 1;
  border-radius: 5px 5px 0 0;
  -moz-border-radius: 5px 5px 0 0;
  -webkit-border-radius: 5px 5px 0 0;
  background: #141414;
  background: -moz-linear-gradient(top, #333333 0%, #141414 100%);
  background: -webkit-gradient(linear, left top, left bottom,
  color-stop(0%, #333333), color-stop(100%, #141414));
  background: -webkit-linear-gradient(top, #333333 0%, #141414 100%);
  background: -o-linear-gradient(top, #333333 0%, #141414 100%);
  background: -ms-linear-gradient(top, #333333 0%, #141414 100%);
  background: linear-gradient(to bottom, #333333 0%, #141414 100%);
}
```

4.4 Index template

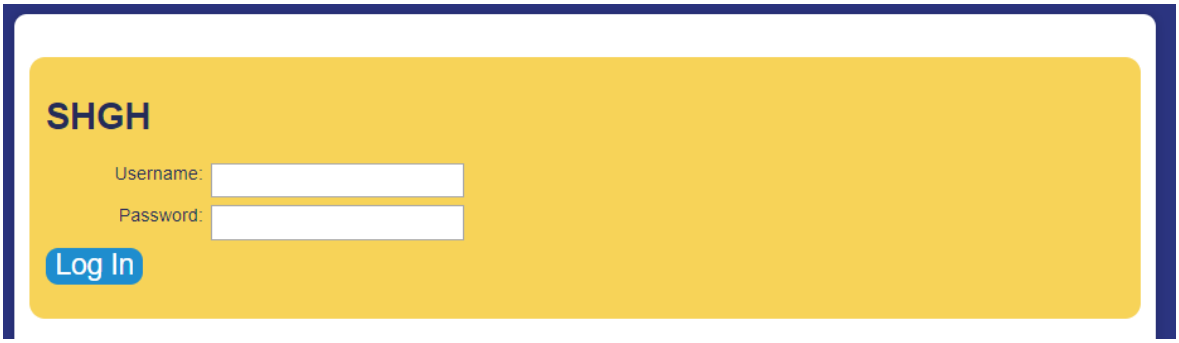

Figure 4.5: Login form in the Index template

The index template is the first one that we will see. It is just a form where the user must write his username and password.

About the design, it is defined in the CSS file:

```
1.  # shgh.css
2.  #orange {
3.      position: relative;
4.      background: #F7D358;
5.      color: #242C58;
6.      font-size: 80%;
7.      margin-top: 10px;
8.      margin-bottom: 10px;
9.      margin-left: 10px;
10.     margin-right: 10px;
11.     padding: 10px;
12.     -webkit-border-radius: 10px;
13.     -moz-border-radius: 10px;
14. }
15. #submit {
16.     background-color: #1E8DCE;
17.     -moz-border-radius: 10px;
18.     -webkit-border-radius: 10px;
19.     border-radius:8px;
20.     color: #fff;
21.     font-family: 'Arial';
22.     font-size: 18px;
23.     text-decoration: none;
24.     cursor: pointer;
25.     border:none;
26. }
27. #submit:hover {
28.     border: none;
29.     background: #ccc;
30.     box-shadow: 0px 0px 1px #777;
31. }
```

#Orange defines the orange square where all the text is shown and #submit defines the design of the button (it will be used in the other templates too).

4.5 Home template

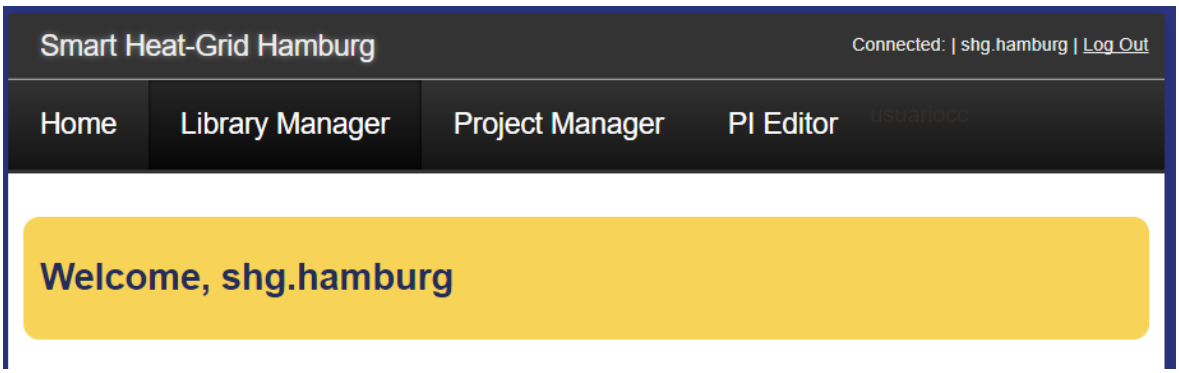

Figure 4.6: Home template

If the login is correct, the user will see the Home template which is just a welcome message but there base template is already visible here. From here the user can access to all the main functions of the application.

4.6 URLs file

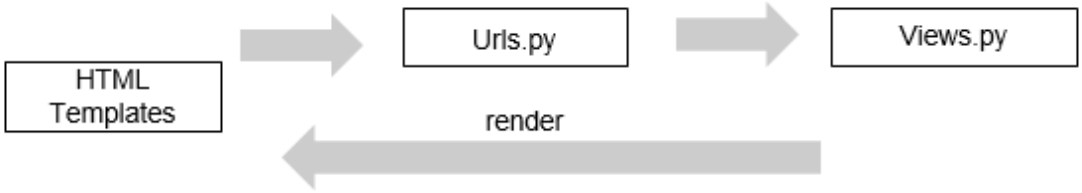

Figure 4.7: Schema of the connection between the modules Url and Views and the HTML templates

The „urls.py“ file is where all the functions of the „views.py“ are called. This is why is an important file in a Django project and it is considered as part of the controller along with the „views.py“ file.

After finishing this thesis, I had a quite long „urls.py“ file because many functions are needed, specially because I am not working with the „models.py“ file which must make the work much easier.

```
1.  # urls.py
2.  urlpatterns = [
3.  url(r'^$', views.index, name='index'),
4.  url(r'^login', views.login, name='login'), url(r'^add/$', views.add, name='add'),

5.  url(r'^add_type/$', views.addType, name='add_type'),
6.  url(r'^add_quantity/$', views.addQuantity, name='add_quantity'),
7.  url(r'^add_parameter/$', views.addParameter, name='add_parameter'),
8.  url(r'^add_method/$', views.addMethod, name='add_method'),
9.  url(r'^select/$', views.select, name='select'),
```

```
10. url(r'^uploads/', views.upload_file, name="uploads"),
11. url(r'^save_component/$', views.saveComponent, name='save_component'),
12. url(r'^create_component/$', views.createComponent, name='create_component'),
13. url(r'^delete_component/$', views.deleteComponent, name='delete_component'),
14. url(r'^select_component/(?P<components>[\w-
    ]+)/$', views.selectComp, name='select_component'),
15. url(r'^select_type/(?P<type>[\w-]+)/$', views.selectType, name='select_type'),
16. url(r'^select_typelib/(?P<type>[\w-
    ]+)/$', views.selectTypeLib, name='select_typelib'),
17. url(r'^select_quantitylib/(?P<quantity>[\w-
    ]+)/$', views.selectQuantityLib, name='select_quantitylib'),
18. url(r'^select_methodLib/(?P<method>[\w-
    ]+)/$', views.selectMethodLib, name='select_methodlib'),
19. url(r'^change_type/(?P<type>[\w-]+)/$', views.changeType, name='change_type'),
20. url(r'^change_quantity/(?P<quantity>[\w-
    ]+)/$', views.changeQuantity, name='change_quantity'),
21. url(r'^change_method/(?P<method>[\w-
    ]+)/$', views.changeMethod, name='change_method'),
22. url(r'^change_folder/(?P<folder>[\w-
    ]+)/$', views.changeFolder, name='change_folder'),
23. url(r'^change_project/(?P<project>[\w-
    ]+)/$', views.changeProject, name='change_project'),
24. url(r'^change_variant/(?P<variant>[\w-
    ]+)/$', views.changeVariant, name='change_variant'),
25. url(r'^delete_type/(?P<type>[\w-]+)/$', views.deleteType, name='delete_type'),
26. url(r'^delete_quantity/(?P<quantity>[\w-
    ]+)/$', views.deleteQuantity, name='delete_quantity'),
27. url(r'^delete_method/(?P<method>[\w-
    ]+)/$', views.deleteMethod, name='delete_method'),
28. url(r'^select_project/(?P<project>[\w-
    ]+)/$', views.select_project, name='select_project'),
29. url(r'^select_project_pm/(?P<project>[\w-
    ]+)/$', views.select_project_pm, name='select_project_pm'),
30. url(r'^select_projectforVariant/(?P<project>[\w-
    ]+)/$', views.selectProjectforVariant, name='select_projectforVariant'),
31. url(r'^select_folder/(?P<folder>[\w-
    ]+)/$', views.select_folder, name='select_folder'),
32. url(r'^select_folder_pm/(?P<folder>[\w-
    ]+)/$', views.select_folder_pm, name='select_folder_pm'),
33. url(r'^select_folderforProject/(?P<folder>[\w-
    ]+)/$', views.selectFolderforProject, name='select_folderforProject'),
34. url(r'^select_variant/(?P<variant>[\w-
    ]+)/$', views.select_variant, name='select_variant'),
35. url(r'^delete_folder/(?P<folder>[\w-
    ]+)/$', views.deleteFolder, name='delete_folder'),
36. url(r'^delete_project/(?P<project>[\w-
    ]+)/$', views.deleteProject, name='delete_project'),
37. url(r'^delete_variant/(?P<variant>[\w-
    ]+)/$', views.deleteVariant, name='delete_variant'),
38. url(r'^pi_editor/$', views.pi_editor, name='pi_editor'),
39. url(r'^libmanager/$', views.libmanager, name='libmanager'),
40. url(r'^libmanager_quantities/$', views.libmanager_quantities, name='libmanager_qu
    antities'),
41. url(r'^libmanager_types/$', views.libmanager_types, name='libmanager_types'),
42. url(r'^libmanager_methods/$', views.libmanager_methods, name='libmanager_methods'
    ),
43. url(r'^pmanager/$', views.pmanager, name='pmanager'),
44. url(r'^simple_upload/$', views.simple_upload, name='upload'),
45. ]
```

Some URLs have an attribute at the end (`(?P<attribute>[\w`) for getting a variable directly from the HTML template without any form needed. When the function is written, we should define it with two variables: request and the attribute, which we can use in the function.

5 Functions

The main functions of the application are located in the Library Manager, the Project Manager and the PI Editor.

5.1 Library Manager

One of the points of this project was to create a Library Manager where the research team can manage and set heat component types with their calculation methods and parameters, e.g. a heat pump that they want to simulate. These are defined as tables in the database:

- ➔ Types: (Heat system) components with an image
- ➔ Quantities: Quantity variables with a unit, type of data and color.
- ➔ Methods: parts of code that are related to a certain type.

The Library Manager will display different options depending on which data we send when we click the link.

First, we can decide if we want to manage the types, quantities or methods.

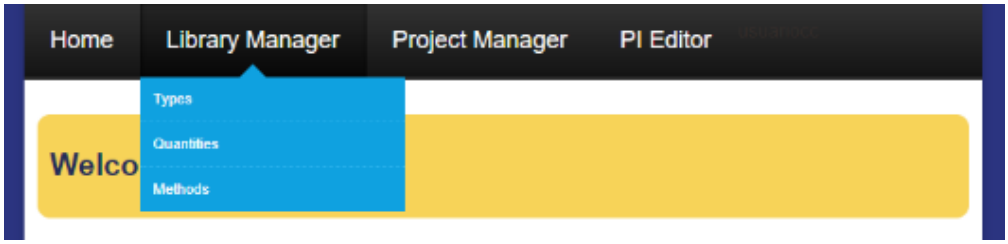

Figure 5.1: Library Manager dropdown menu

5.1.1 Types

If we click the „types“ button, we are calling the function „libmanager_types“ that opens the libmanager but only the options related to types will be shown:

```
# views.py
def libmanager_types(request):
    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn,  dbdataTimeZone="CET")
    myDbConnector.connect()
    global type_data
    type_data = myDbConnector.showTypes()

    form = CreateType()
    context = {'names': type_data,
               'username': name_conn,
                'form': form,
              }
    return render(request, 'shgh/libmanager.html', context)
```

The library manager will show the list of types that exist in the database and some options concerning them (we can create a new type, or update/delete an existing one). This is why we have to call the function „showTypes“ in the „dbPointer“:

```
# dbPointer.py
def showTypes(self):
    self.conn.set_isolation_level(0)
    # try:
    print("The following types exist: ")
    self.cur.execute("SELECT type_name FROM smardism_libraries_db_dev_pj.types;")

    row = self.cur.fetchone()

    list = []
    while row is not None:
        rows1=str.encode(row[0])
        rows = rows1.decode("utf-8").replace(' ', '_')
        list.append(rows)
        print (list)
        row = self.cur.fetchone()
    return (list)
```

It selects all the types in the schema „smardism_libraries_db_dev_pj“ and appends them into a list where we do certain changes in order that the Python functions can read correctly all the data.

The Library Manager will be opened and let us create a new type or choose one of the types that we have in the whole database for checking which data is stored in that row, having the possibility to update or delete it.

Figure 5.2: Library Manager screen with types

Create Type

When creating a new type, the user sends the chosen name and image to the function „addType“:

```
1.  # libmanager.html
2.  {% if names %}
3.      <form class="formhorizontal alinear" action='{% url 'add_type' %}' method=
        "post" enctype="multipart/form-data"}>
4.          {% csrf_token %}
5.          {{ form.as_p }}
6.
7.          {% csrf_token %}
8.          <input type="file" name="myfile" accept="image/*">
9.          <br><input type="submit" value="Create" id="submit"/>
10.     </form>
11.  {% endif %}
```

In this code, „names“ is how I named the types that are sent through the „libmanager_types“ function. Then there is a form where the user writes the name of the type (this text field is provided by the form CreateType) and uploads an image.

The type name is included inside of the form that we are calling in this template and that we receive from the function. On the other side, the image url will be the route to the static directory where the uploaded images are stored.

If the formulary is correct, this data will be sent to the function „add_type“:

```
1.  # views.py
2.  def addType(request):
3.
4.      form = CreateType(request.POST, request.FILES)
5.
6.      if form.is_valid():  #if the user is sending the form with the correct data
7.
8.          data = form.cleaned_data
9.          type_name = data.get("type_name")
10.
11.         # for the image field
```

```
        myfile = request.FILES['myfile']
        fs = FileSystemStorage()
        filename = fs.save(myfile.name, myfile)
        image_url = fs.url(filename)

        #Database connection through db_pointer.py
        myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn,
                                        dbdataTimeZone="CET")
        myDbConnector.connect()
        myDbConnector.createTypesSimple(type_name, image_url)

    return render(request, 'shgh/libmanager.html', {'form': form})
```

It gets the type name and the image url and calls the function „createTypesSimple“ from the dbPointer:

```
# dbPointer.py
def createTypesSimple(self, name, image):
    type_name = name
    self.dbTable = "smardism_libraries_db_dev_pj.types"
    imageplaceholder = image
    try:
        self.cur.execute(
            "INSERT INTO {0}(type_name, type_date, image_url) VALUES ('{1}',
             transaction_timestamp() AT TIME ZONE '{2}', '{3}');"
                .format(self.dbTable, type_name, self.dataTimeZone,
                 imageplaceholder))
        print("Type {0} created.".format(type_name))
    except:
        self.cur.execute(
            "INSERT INTO {0}(type_name, type_date, image_url) VALUES ('{1}',
             transaction_timestamp() AT TIME ZONE '{2}', '{3}')"
            " ON CONFLICT DO NOTHING;".format(self.dbTable, type_name,
            self.dataTimeZone, imageplaceholder))
        print("Type {0} loaded.".format(type_name))
    self.conn.commit()
```

Here, the data gets stored in the table „Types“ from the schema „smardism_libraries_db_dev_pj“ if the name is not duplicated. If it is, the function will do nothing in the database.

The type table has three different columns, where one of them (type_date) can be set through the „db_pointer.py“ just saving the current date when the type is created.

Select Type

When the user selects one type of the dropdown menu the page will refresh because we are calling another function called „select_typelib“.

```
# libmanager.html
{% if names %}
   <ul>
      <select name="select" onchange="location.href=this.options[this.selected
       Index].value" >
         <option type="text" value="0" id="text">Type Name</option>
         {% for type in names %}
         <option type="text" id="type" value="{% url 'select_typelib' type=
          type %}"> {{ type }}</option>
         {% endfor %}
```

```
        </select>
    </ul>
{% endif %}
```

Then we will have two new menus on the same template. In both menus, the data stored in this selected type (type name and image) will apear thanks to the function „select_typelib“.

```
# views.py
def selectTypeLib(request, type):

    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
    myDbConnector.connect()

    form = CreateType()
    image_url = myDbConnector.showImage(type)
    context = {
        'typeLib': type,
        'names': type_data,
        'image_url': image_url,
        'form': form,
        'username': name_conn,
    }
    return render(request, 'shgh/libmanager.html', context)
```

On the left we can delete all the data concerning this type and on the right we can update the fields we choose.

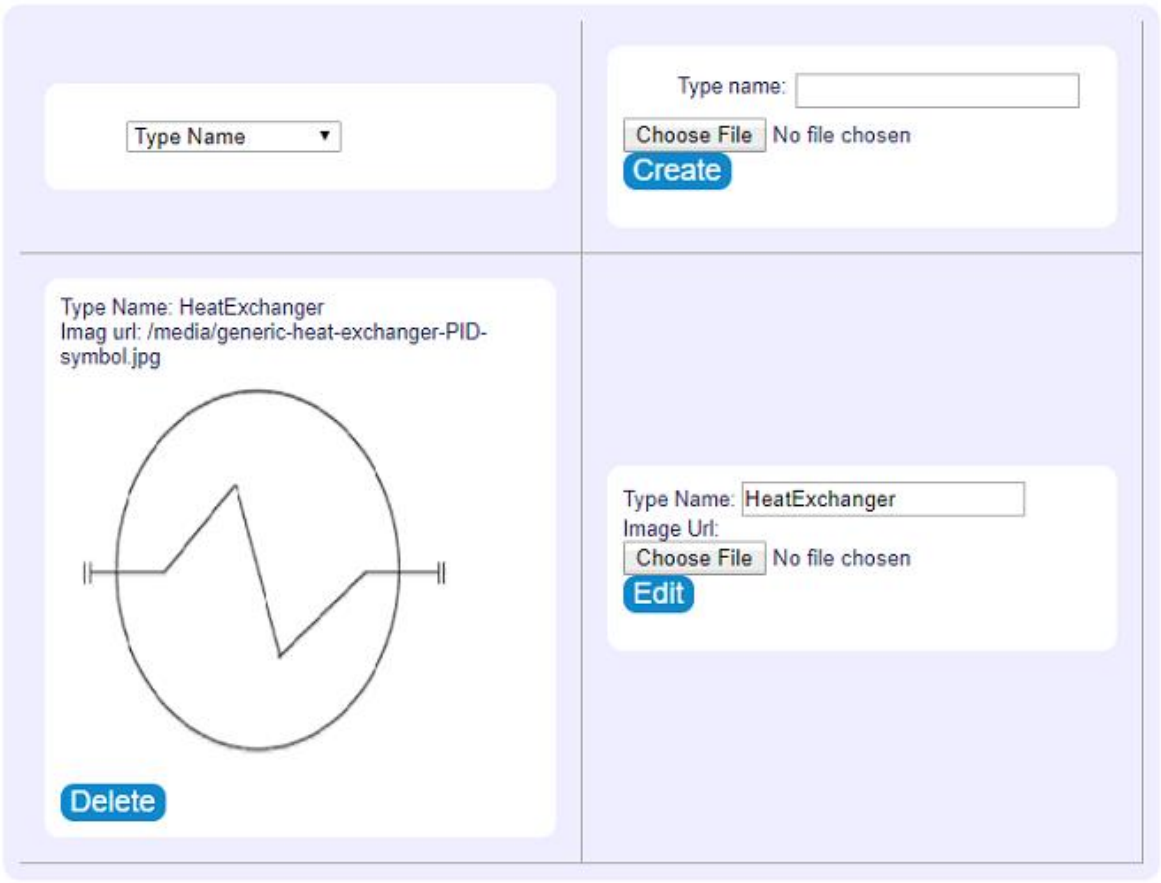

Figure 5.3: Library Manager with types when selecting a type

But first of all we should be aware about the different dependencies between tables before doing any change in them. In this case the type name is also referenced in the table methods as a foreign key, so if we delete or update one type we must define in the pgAdmin what action will do a method that depends on this type:

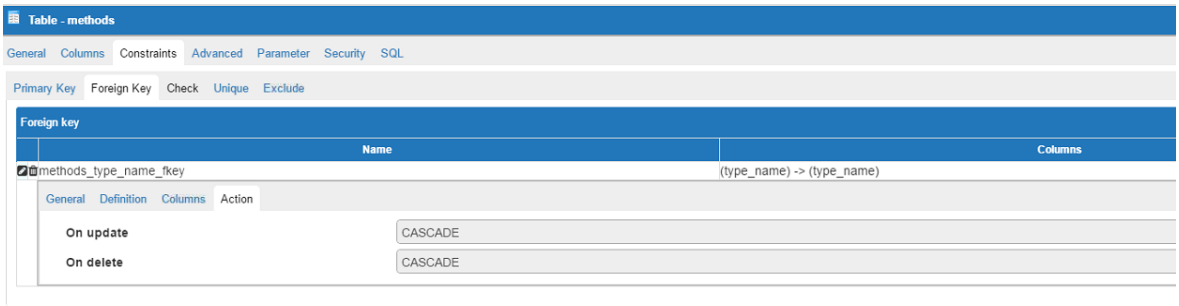

Figure 5.4: Setting the action "CASCADE" when a foreign key of "methods" is updated/deleted

If we select „CASCADE", now if we delete a type, all the methods with this type referenced as a foreign key will be automatically deleted.

The same must be done with the components, a table of another schema that also depends on the types.

Update Type

One of the options that appear after selecting a type is updating it.

```
1.  # libmanager.html
2.  <form class="form-horizontal alinear"
    action='{% url 'change_type' type=typeLib %}' method="post"
    enctype="multipart/form-data"}>
3.      {% csrf_token %}
4.      Type Name: <input type="text" name="newname" value = "{{ typeLib }}">
5.      <br>Image Url: <input type="file" name="myfile" accept="image/*">
6.      <br><input type="submit" value="Edit" id="submit"/>
7.  </form>
```

The template shows the current data for this type in two inputs and give us the option to edit them through the function „change_type":

```
1.  # views.py
2.  def changeType(request, type):
3.      myfile = request.FILES['myfile']
4.      fs = FileSystemStorage()
5.      filename = fs.save(myfile.name, myfile)
6.      image_url = fs.url(filename)
7.      newname = request.POST['newname']
8.
9.      myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
10.     myDbConnector.connect()
11.     image_url_old = myDbConnector.showImage(type)
12.     myDbConnector.changeType(type, image_url_old, image_url, newname)
13.     return render(request, 'shgh/libmanager.html')
```

This function saves the new data and also requests the old one to the database because it is needed to update the row. It is the same request that is used for selecting and showing the data of the type. With PostgreSQL we have to define which data we want to introduce and which one will be replaced.

```
# dbpointer.py
def changeType(self, name, image_old, image, newname):
    self.dbTable = "smardism_libraries_db_dev_pj.types"
    self.cur.execute(
        "UPDATE {0}  SET image_url='{1}' WHERE image_url='{2}';
         UPDATE {0} SET type_name='{3}' WHERE type_name='{4}';"
        .format(self.dbTable, image, image_old, newname, name))
    self.conn.commit()
```

Delete Type

The other option that we have is to delete a type. In the template we can see the name of the type and also the image, which is taken using the „img src“ command because in the database we only have the url of the image but not the image itself. The static files must be loaded aswell.

```
# libmanager.html
Type Name: {{ typeLib }}
<br>Imag url: {{ image_url }}
{% load static %}
<br><img src="{% static image_url %}" alt="My image" style= "height: 250px;
width: 250px;"/>
<br><a  href="{% url 'delete_type' type=typeLib%}">
<input type="submit" value="Delete" id="submit"/></a>
```

If this is the type that we want to delete of the database, we just click the button „Delete“ and a function will be called:

```
# views.py
def deleteType(request, type):
    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser="shg.hamburg", dbPw="shgh#", dbdataTimeZone="CET")
    myDbConnector.connect()
    myDbConnector.deleteType(type)

    return render(request, 'shgh/libmanager.html')
```

For deleting the type, we open this function in the dbPointer:

```
# dbpointer.py
def deleteType(self, type):
    self.dbTable = "smardism_libraries_db_dev_pj.types"
    self.cur.execute(
        "DELETE FROM {0}  WHERE type_name='{1}';".format(self.dbTable, type))
    self.conn.commit()
```

We just have to render the name of the table where there is the row that we want to delete and the type name (as it is the Primary Key [19]) of that row.

5.1.2 Quantities

The different functions for the quantities table are easier to implement because we do not have, for now, any dependence with other tables and all the variables are just text fields. So it is just needed to get the data from the form where the user is writting it.

When the function renders the „libmanager“ template, in this case it sends the data of the quantities instead of the data of the types. Thus, the design of the web page will look slightly different.

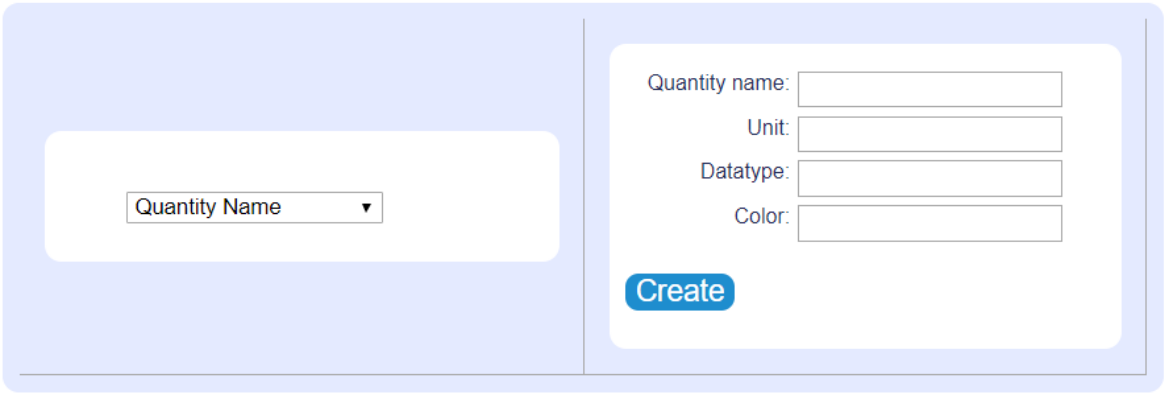

Figure 5.5: Library Manager screen with quantities

But as in general all the process is very similar for the three tables of the libraries and I explained it at the previous section (Types), here I will only give the information that can be a bit different.

Create Quantities

When creating a new quantity, the user sends the chosen name, unit, datatype and color to the function „addQuantity“. These four variables are defined in the form „CreateQuantity“:

```
1.  # forms.py
2.  class CreateQuantity(forms.Form):
3.      quantity_name = forms.CharField(widget=forms.TextInput(), required=True)
4.      unit = forms.CharField(widget=forms.TextInput(), required=True)
5.      datatype = forms.CharField(widget=forms.TextInput(), required=True)
6.      color = forms.CharField(widget=forms.TextInput(), required=True)
```

If the form is completed correctly, this data will be sent to the function „addQuantity“ and it will be stored through the „dbPointer“, just as it happens when creating a type. In the dbPointer the data gets stored in the table „Quantities“ from the schema „smardism_libraries_db_dev_pj“ if the data is not duplicated.

Select Quantity

When the user selects one quantity of the dropdown menu the page will refresh and it looks like this with the two new menus:

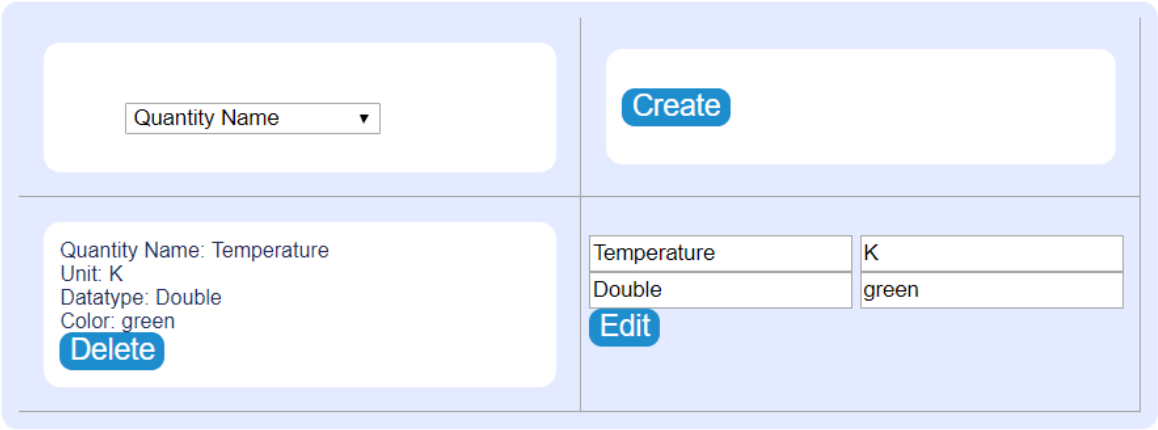

Figure 5.6: Library Manager screen with quantities when selecting a quantity

The process for **update** and **delete** is like for the types but with the fact that we only have text fields and that we need to call three functions („showUnit“, „showDataType“ and „showColor“) instead of one („showImage“) for getting the current data that we want to update from the database or just show when selecting a quantity.

```
1.  # views.py
2.  def changeQuantity(request, quantity):
3.
4.      …
5.      unit_old = myDbConnector.showUnit(quantity)
6.      datatype_old = myDbConnector.showDataType(quantity)
7.      color_old = myDbConnector.showColor(quantity)
8.      …
9.      return render(request, 'shgh/libmanager.html')
```

5.1.3 Methods

The methods are the codes that we introduce to a type for changing something. If we click the button „Methods“ in the dropdown menu of the Library Manager, we are calling the function „libmanager_methods“:

```
1.  # views.py
2.  def libmanager_methods(request):
3.
4.      myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
5.      myDbConnector.connect()
6.
7.      global method_data
8.      method_data = myDbConnector.showMethods()
9.
10.     type_data = myDbConnector.showTypes()
11.     context = {
12.                 'types': type_data,
13.                  'username': name_conn,
14.                 'methods': method_data}
15.     return render (request,'shgh/libmanager.html', context)
```

In this case, for opening the template we have to request the list of types to the database with the function „showTypes“. This is because the type name is one of the variables that a method has, so it is needed to select one type in order to create a new method.

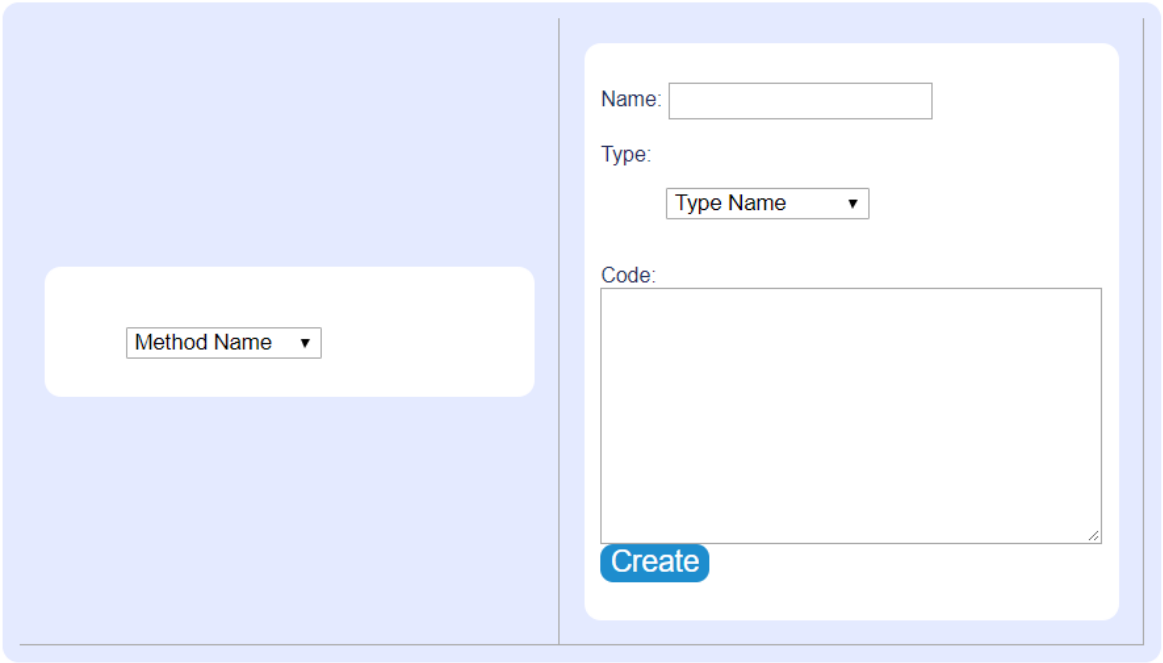

Figure 5.7: Library Manager screen with methods

Create Method

When creating a new method, the user send the chosen name, the code and the type that he has to select from the dropdown menu. The user has to select one type there in order to call the function „select_type“, which is required for storing one method in

relation to a specific type. This is the main difference that we have with the other two tables and this is why the HTML code is quite longer.

```
# libmanager.html
{% if methods %}
   <form class="form-horizontal alinear"
   action='{% url 'add_method' %}' method="post"}>
       {% csrf_token %}
       <br>Name: <input id="method_name" type="text" name="method_name" >
       <br><br>Type:
       <ul>
            {% if current_typeforMethod %}
                <select name="dropdown"  >
                     <option value="{% url 'select_type'
                     type=current_typeforMethod %}">{{ current_typeforMethod }}
                     </option>
                     {% for type in types %}
                     <option value="{% url 'select_type' type=type %}">
                     {{ type }}</option>
                     {% endfor %}
               </select>

            {% else %}
               <select name="dropdown"
              onchange="location.href=this.options[this.selectedIndex].value" >
                   <option value="0">Type Name</option>
                   {% for type in types %}
                   <option value="{% url 'select_type' type=type %}">
                   {{ type }}</option>
                   {% endfor %}
               </select>
            {% endif %}
        </ul>
        <br>Code:  <textarea id="method_code" name="method_code"
        rows="10" cols="40">
```

The method code is usually a long text, this is why it is used a text area instead of a text input (in a text area we can select the rows and columns of the input).

```
# views.py
def addMethod(request):
    form = CreateMethod(request.POST or None)

    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
    myDbConnector.connect()

    if form.is_valid():
        data = form.cleaned_data
        method_name = data.get("method_name")
        method_code = request.POST["method_code"]
        type_name = current_typeforMethod

        myDbConnector.createMethods(method_name, method_code, type_name)

    return render(request, 'shgh/libmanager.html')
```

For the method code we must get the data from the form with the request.POST command and the type name is chosen before from the dropdownmenu and stored in a global variable before calling this „addMethod“ function.

The „createMethods“ function is not so complex and is similar to the other ones („createTypes“ and „createQuantities“).

Here the data gets stored in the table „Methods“ from the schema „smardism_libraries_db_dev_pj“ if the data is not duplicated.

Select Method

When the user selects one method this function is called:

```
# views.py
def selectMethodLib(request, method):
    …

    method_code = myDbConnector.showCode(method)
    typefromMethod = myDbConnector.showTypefromMethod(method)
    …
    return render(request, 'shgh/libmanager.html', context)
```

In this case two functions are needed to get the current data (code and typename, both taken from the table „methods“).

The process for **update** and **delete** is like for the types and quantities but in this case we need to call three functions („showCode“, „showTypefromMethod“ as shown in the previous functon.

5.2 Project Manager

The project manager is similar to the library manager, but in this case we are creating, updating and deleting folders, projects and variants. These tables have more dependencies between them because a variant belongs to a project and a folder, and a project belongs to a folder. Because of this, I implemented them in the same screen.

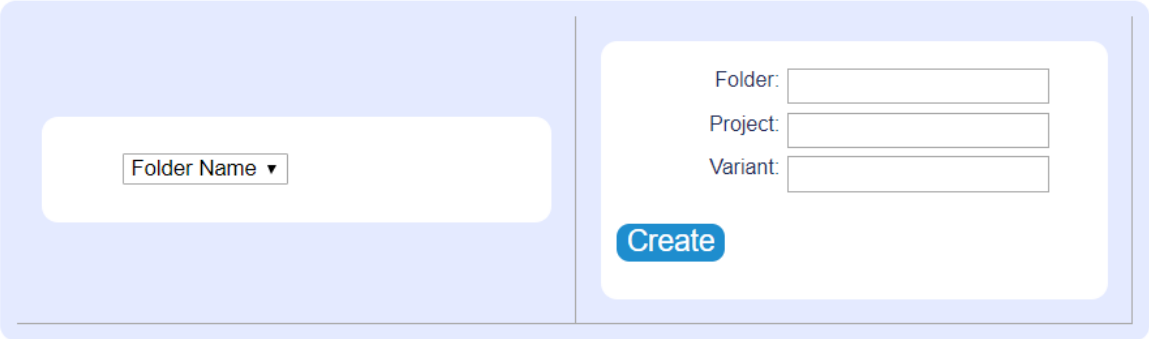

Figure 5.8: Project Manager screen

When we click on the „Project Manager“ button of the menu, we call the function „pmanager“ that renders this template.

```
# views.py
def pmanager(request):
    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
    myDbConnector.connect()
    form = CreateProject()

    global folder_data
    folder_data = myDbConnector.showFolders()
    context = {
        'username': name_conn,
        'names': folder_data,
        'form': form,
    }
    return render (request,'shgh/pmanager.html', context)
```

It requests to the database the different stored folders and then we can show them through a dropdown menu in the template.

```
# dbPointer.py
def showFolders(self):
    self.conn.set_isolation_level(0)
    print("The following folders exist: ")
    self.cur.execute("SELECT folder_name FROM smardism_project_db_dev.folders;")
    row = self.cur.fetchone()

    list = []
    while row is not None:
        rows1 = str.encode(row[0])
        rows = rows1.decode("utf-8").replace(' ', '_')
        list.append(rows)
        row = self.cur.fetchone()
    return (list)
```

In this function we select all the existing folders in the schema „smardism_project_db_dev“ and we append each one in a list.

Create

The function for creating the three rows in the database is the same, but if for example a duplicate folder is detected, this existing folder will be loaded (not created again) and the project will be created in this selected folder.

```
# pmanager.html
<form class="form-horizontal alinear" action='{% url 'add' %}' method="post"}>
    {% csrf_token %}
    {{ form.as_p }}
    <br><input type="submit" value="Create" id="submit"/>
</form>
```

In the form we have the primary keys [19] of the three tables because they are the only ones that the user has to give:

```
# forms.py
class CreateProject(forms.Form):

    folder = forms.CharField(widget=forms.TextInput(), required=True)
    project = forms.CharField(widget=forms.TextInput(), required=True)
    variant = forms.CharField(widget=forms.TextInput(), required=True)
```

If the form is completed correctly, this data will be sent to the function „add“:

```
# views.py
def add(request):
    form = CreateProject(request.POST or None)

    if form.is_valid():
        data = form.cleaned_data
        folder = data.get("folder")
        project = data.get("project")
        variant = data.get("variant")
        myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name_conn, dbPw=password_conn,dbdataTimeZone="CET")
        myDbConnector.connect()
        myDbConnector.createProjects(folder, project, variant)

        global folder_data
        folder_data = myDbConnector.showFolders()
        context= { 'username': name_conn,
               'form': form,
               'names': folder_data,
        }

    return render(request, 'shgh/pmanager.html', context)
```

We can see that for now this process is similar to each one of the Library Manager. But the main difference will be in the dbPointer function, because it has to do something in three tables in the same function.

```
# dbPointer.py
def createProjects(self, folder, project, variant):
    self.conn.set_isolation_level(0)
    folder_name = folder
```

```
5.     self.dbTable = "smardism_project_db_dev.folders"
6.     try:
7.        self.cur.execute("INSERT INTO {0}(folder_name, date, folder_owner)
          VALUES ('{1}', transaction_timestamp() AT TIME ZONE '{2}', '{3}');"
          .format(self.dbTable, folder_name, self.dataTimeZone, self.dbUser))
8.        print("Folder {0} created.".format(folder_name))
9.      except:
10.       self.cur.execute("INSERT INTO {0}(folder_name, date, folder_owner)
          VALUES ('{1}', transaction_timestamp() AT TIME ZONE '{2}', '{3}')
          ON CONFLICT DO NOTHING;".format(self.dbTable, folder_name,
          self.dataTimeZone, self.dbUser))
11.        print("Folder {0} loaded.".format(folder_name))
12.
13.        self.conn.commit()
14.
15.        print("Project creation started.")
16.        project_name = project
17.        self.dbTable = "smardism_project_db_dev.projects"
18.        try:
19.            self.conn.set_isolation_level(0)
20.            self.cur.execute(
21.                "INSERT INTO {0}(project_name, date, project_owner, folder_name)
                   VALUES ('{1}', transaction_timestamp() AT TIME ZONE '{2}',
                   '{3}', '{4}');".format(
22.                self.dbTable, project_name,self.dataTimeZone, self.dbUser,
                   folder_name))
23.            print("Project {0} created.".format(project_name))
24.        except:
25.            self.cur.execute(
26.                "INSERT INTO {0}(project_name, date, project_owner, folder_name)
                   VALUES ('{1}', transaction_timestamp() AT TIME ZONE '{2}',
                   '{3}', '{4}') ON CONFLICT DO NOTHING;".format(
27.                self.dbTable, project_name, self.dataTimeZone, self.dbUser,
                   folder_name))
28.            print("Project {0} loaded.".format(project_name))
29.        self.conn.commit()
30.
31.        print("Variant creation started. ")
32.        variant_name = variant
33.        self.dbTable = "smardism_project_db_dev.variants"
34.        try:
35.            self.cur.execute(
36.                "INSERT INTO {0}(variant_name, date, project_name, folder_name,
                   variant_owner) VALUES ('{1}', transaction_timestamp()
                   AT TIME ZONE '{2}', '{3}', '{4}', '{5}');".format(
37.                self.dbTable, variant_name, self.dataTimeZone, project_name,
                   folder_name, self.dbUser))
38.            print("Variant {0} created.".format(variant_name))
39.        except:
40.            self.cur.execute(
41.                "INSERT INTO {0}(variant_name, date, project_name, folder_name,
                   variant_owner) VALUES ('{1}', transaction_timestamp()
                   AT TIME ZONE '{2}', '{3}', '{4}', '{5}') ON CONFLICT DO NOTHING;"
                   .format(
42.                self.dbTable, variant_name, self.dataTimeZone, project_name,
                   folder_name, self.dbUser))
43.            print("Variant {0} loaded.".format(variant_name))
44.        self.conn.commit()
```

First it tries to create a new folder. If the data provided by the user is duplicated (the folder already exists), the existing folder will be loaded and the project will be created there (the existing folder will be the foreign key [20] of this new project). And the same process with the project and the variant.

Select Folder

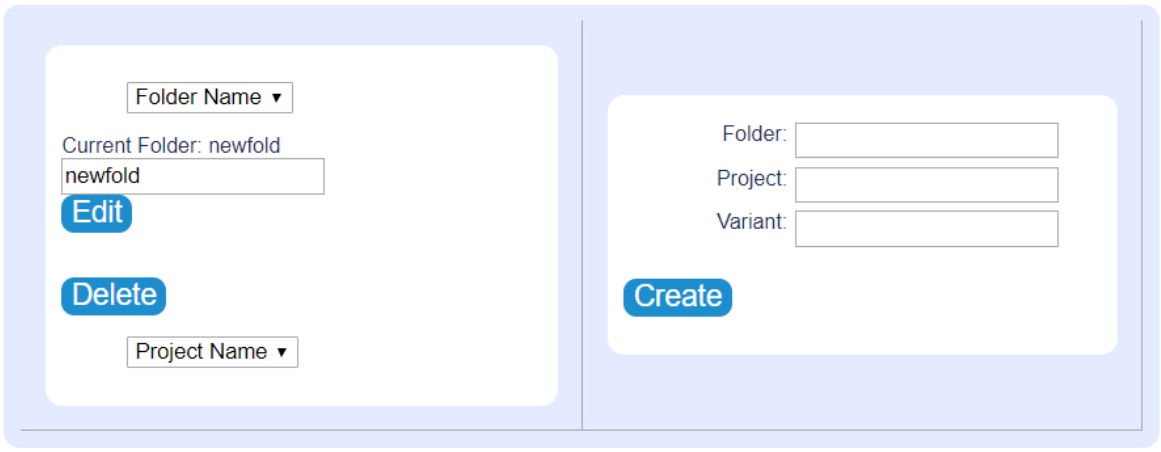

Figure 5.9: Project Manager screen when a folder is selected

When a folder is selected we can edit or delete it but also a new dropdown menu will appear in order to select one project in the folder. When we select a project it will happen the same (we can edit and update it, and also select a variant in the project).

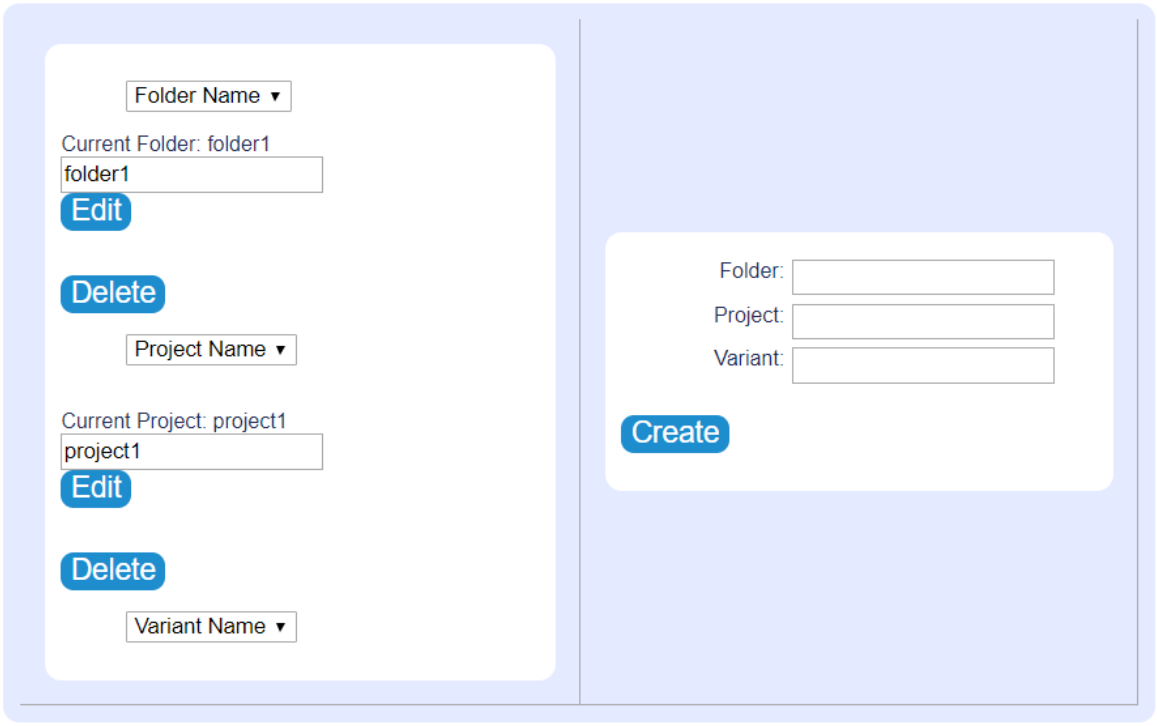

Figure 5.10: Project Manager screen when a project is selected

The functions for updating are like this:

```
1.  # views.py
2.  def changeFolder(request, folder):
3.      newname = request.POST['newfolder']
```

```
4.
5.      myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
6.      myDbConnector.connect()
7.      myDbConnector.changeFolder(folder, newname)
8.
9.      return render(request, 'shgh/pmanager.html')
10.
11. def changeFolder(request, folder):
12.     newname = request.POST['newfolder']
13.
14.     myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
15.     myDbConnector.connect()
16.     myDbConnector.changeFolder(folder, newname)
17.
18.     return render(request, 'shgh/pmanager.html')
19.
20. def changeFolder(request, folder):
21.     newname = request.POST['newfolder']
22.
23.     myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
24.     myDbConnector.connect()
25.     myDbConnector.changeFolder(folder, newname)
26.
27.     return render(request, 'shgh/pmanager.html')
```

For the project we have one foreign key and for the variant we have two, so we must define it for updating the tables.

The codes for deleting are the following:

```
1.  # dbPointer.py
2.  def deleteFolder(request, folder):
3.      myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
4.      myDbConnector.connect()
5.      myDbConnector.deleteFolder(folder)
6.
7.      return render(request, 'shgh/pmanager.html')
8.
9.  def deleteProject(request, project):
10.     myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
11.     myDbConnector.connect()
12.     myDbConnector.deleteProject(project)
13.
14.     return render(request, 'shgh/pmanager.html')
15.
16. def deleteVariant(request, variant):
17.     myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14",
        dbName="SHGH", dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
18.     myDbConnector.connect()
19.     myDbConnector.deleteVariant(variant)
20.
21.     return render(request, 'shgh/pmanager.html')
```

5.3 PI Editor

This is the main part of the web interface and also the most complex, because it is needed to work with javascript in order to create a droppable canvas and different draggable objects.

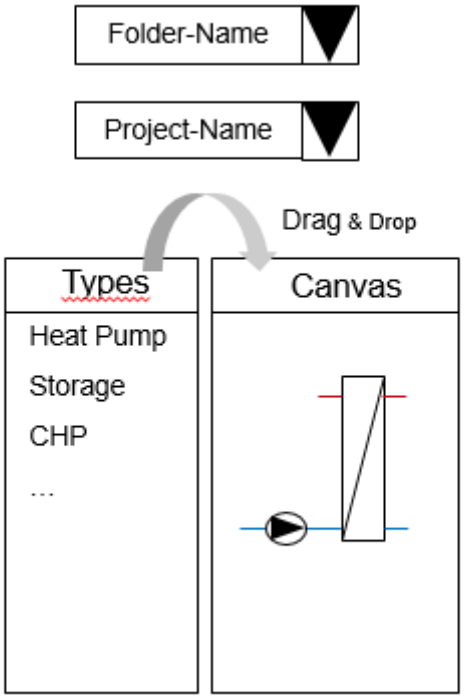

Figure 5.11: schema of the PI Editor

When we click the „PI Editor“ button the screen that appears is very similar to the one of the Project Manager.

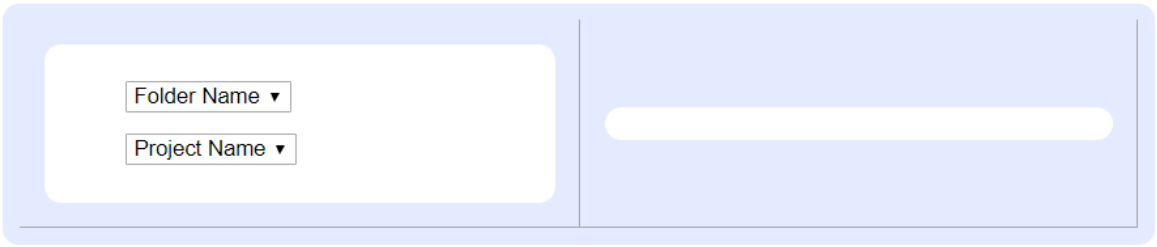

Figure 5.12: PI Editor screen

5.3.1 Use cases

➔ First we have select a folder and then a project for seing which components are created into this project and what is their position in the canvas. Also we will have a column that shows which types are created in the whole table.

```
1.  # views.py
2.  def pim(request):
3.
4.      myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
5.      myDbConnector.connect()
6.
7.      folder_data = myDbConnector.showFolders()
8.
9.      context = {'names': folder_data,
10.                 'username': name_conn,
11.             }
12.     return render(request, 'shgh/pim2backup.html', context)
```

➔ All the different types are represented as draggable objects that can be dropped into the canvas.

➔ When a new draggable type is dropped into the canvas, the Python function „create_component“ is called through a url and some data (the name of the type and the position coordinates) is sent.

➔ The components, which are already in the canvas, they can be dragged aswell into a different position. In this case the function called is „save_component“, that only updates the position of the selected component.

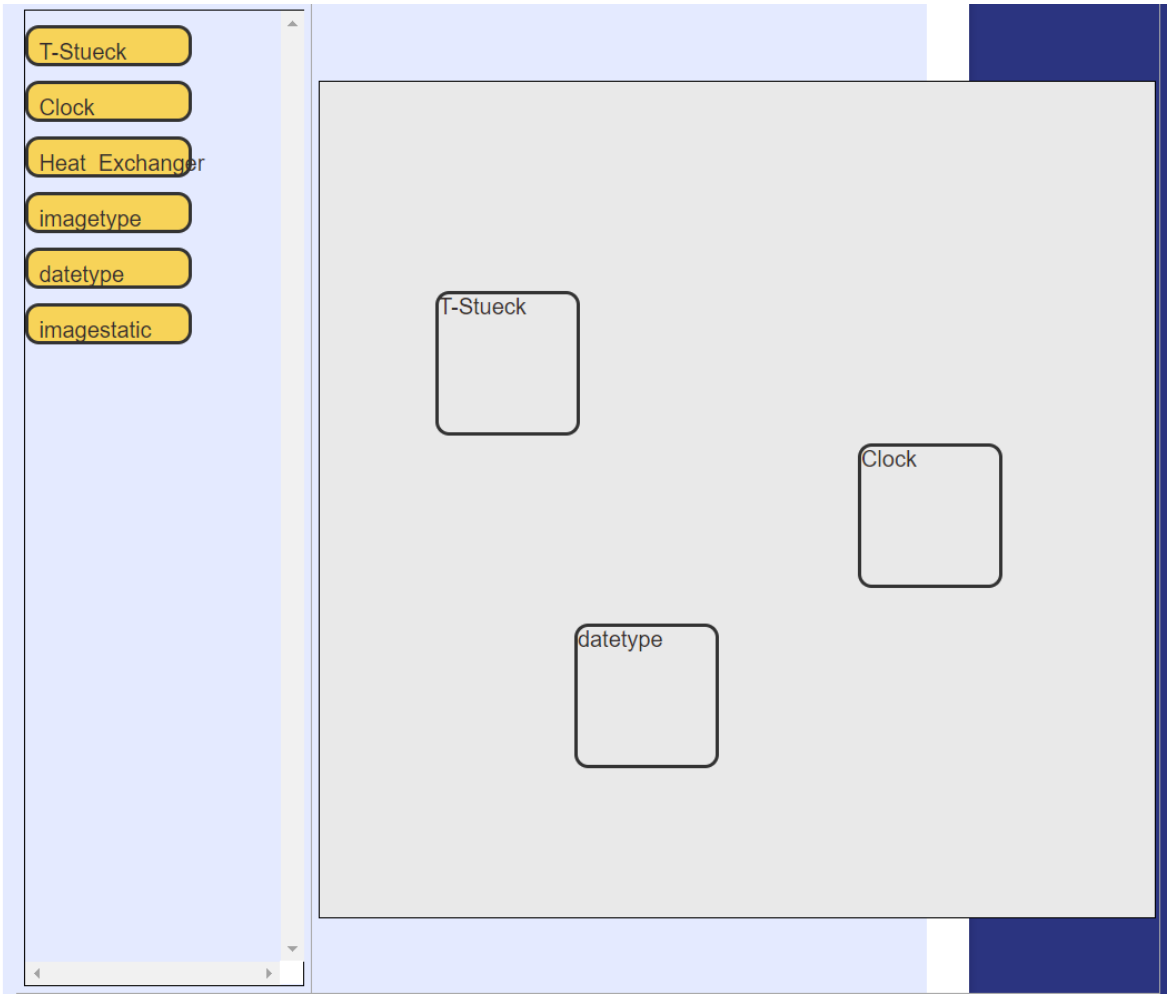

Figure 5.13: PI Editor screen when selecting a project

5.3.2 Drag & Drop

For creating these draggable [28] and droppable [29] objects I needed to work with Javascript and jQuery. Thus, I have a template written in HTML and Javascript.

At first, when we click on the menu to access to the PI Editor, we will see a screen which is quite similar to the Project Manager, but after selecting a folder and a project the canvas appears and also a list of all the types that we have in the database. This canvas can be empty or not depending on if we have already some components dropped there.

```
1.  #views.py
2.  def select_project(request, project):
3.
4.          global project_now
5.          project_now = project.replace('_', ' ')
6.
```

```
        myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
        dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
        myDbConnector.connect()

        types = myDbConnector.showTypes()
        typesfromComp = myDbConnector.showTypesfromComp(folder_now, project)
        component_data = myDbConnector.showComponents(folder_now, project)
        pos_x = myDbConnector.showposX(folder_now, project)
        pos_y = myDbConnector.showposY(folder_now, project)
        image_type=myDbConnector.showImages()
        folder = folder_now
        print(variant_data)
        context = {'names': folder_data,
                    'types': types,
                    'current_folder': folder,
                    'typesfromComp_data': typesfromComp,
                    'component_data': component_data,
                    'current_top': pos_y,
                     'current_left': pos_x,
                    'images': image_type,
                    'projects': project_data,
                    'current_project_space': project_now,
                    'current_project': project,
                    'username': name_conn,
                    }
        return render(request, 'shgh/pim2backup.html', context)
```

As it is shown, the function „select_project“ for the PI Editor is quite more complex than the one for the Project Manage.

The function returns a lot of data that will be used in the template. Thus, it is needed to request all these variables to the database through the dbPointer:

- **Types**: we need to make them appear in the left side of the screen in order that they can be dragged by the user.
- **Types from Component**: this function of the dbPointer takes the „type_name“ variable from the component. As the component is just a type drop into the canvas it is more suitable to show it as a type in the interface.
- **Top and Left**: these are the coordinates X and Y of the component, necessary to show the component in the correct position into the canvas.

We need that the types at the left column and the components at the canvas are draggable.

First we set the CSS style of these items. The types will be stored in a division called #typePile2 and the components in #typePile:

```
# pieditor.html
#droppable { width: 600px; height: 600px; float: left; position:relative }
#typePile2 {
                width:200px;
                height:700px;
                overflow: scroll;
                border:1px solid #000000;
            }
#typePile {
     width: 600px;
      height: 600px;

}
```

```
        #typePile div {
            width: 100px;
            height: 100px;
            border: 2px solid #333;
            -moz-border-radius: 10px;
            -webkit-border-radius: 10px;
            position:absolute;
        }

        #typePile2 div {
            background: #F7D358;
            width: 100px;
            height: 10px;
            padding: 0.5em;
            margin: 10px 10px 10px 0;
            border: 2px solid #333;
            -moz-border-radius: 10px;
            -webkit-border-radius: 10px;
        }
```

After that I opened a script for writing the function init(), that will manage all the Javascript methods for the PI Editor.

First of all, we save the data from the Python function into new variables that we can use in Javascript:

```
# pieditor.html
function init() {

        var posx = [
                {% for left in current_left %}
                "{{left}}",
                {% endfor %}
             ];
        var posy = [
                {% for top in current_top %}
                "{{top}}",
                {% endfor %}
             ];
        var image_url = [
                   {% for image_url in images %}
                "{{image_url}}",
                {% endfor %}
               ];
        var types = [
                {% for types in typesfromComp_data %}
                "{{types}}",
                {% endfor %}
             ];
        var components = [
                {% for components in component_data %}
                "{{components}}",
                {% endfor %}
             ];
        var types2 = [
                {% for types2 in types %}
                "{{types2}}",
                {% endfor %}
             ];
```

Then a loop is created in order to store a new division into the typePile for every component that we have in the selected project.

```
# pieditor.html
for ( var i=0; i<types.length; i++ ) {
        $('<div>' + types[i] + '</div>').data('type', types[i])
        .data('component', components[i]).css({ 'top': posy[i], 'left': posx[i]})
        .attr('id', 'card' + types[i]).appendTo('#typePile').draggable({
            scroll: false,
            containment: '#droppable',
            revert: "invalid",
            position: "absolute",
            cursor: 'move',
            drag: function (event, ui) {
                $("#posx").text(ui.position.left);
                $("#posy").text(ui.position.top);
            },
        });
    }
```

The name of each one will be the type name of the component, because the user will see them as types, but we use the component name in order that the function can know that we are working with a component instead of just a type and the result will be different.

In the same line the position is set as well.

The loop for storing the types is a bit different. Firstly because we set the component data as null (now they are identified as types instead of components) and the other main difference is that in this case we have a helper. It creates a clone that can be dragged multiple times. (there can be many „clones“ into the draggable and we can still create new ones if the user needs the same type several times in the project).

```
# pieditor.html
for ( var i=0; i<types2.length; i++ ) {
        $('<div>' + types2[i] + '</div>').data('type', types2[i])
        .data('component', null).attr('id', 'card' + types2[i])
        .appendTo('#typePile2').draggable({
            scroll: false,
            revert: "invalid",
             position: "absolute",
            cursor: 'move',
            top: '100px',
            zIndex: 1000,
            helper:  'clone',

            drag: function (event, ui) {
                $("#posx").text(ui.position.left);
                $("#posy").text(ui.position.top);

            },
        });
    }
```

Then I created a function for the droppable object, the canvas:

```
# pieditor.html
$( "#droppable" ).droppable({

```

```
4.        accept: "#typePile div, #typePile2 div",
5.        classes: {
6.          "ui-droppable-active": "ui-state-active",
7.          "ui-droppable-hover": "ui-state-hover"
8.        },
9.          position: "relative",
10.
11.         tolerance: "fit",
12.
13.       drop: function( event, ui ) {
14.
15.         // Get the type name:
16.
17.         var TypeName = ui.draggable.data( 'type' );
18.
19.         var CompName = ui.draggable.data( 'component' );
20.
21.         // Get mouse position relative to drop target:
22.         var dropPositionX = event.pageX - $(this).offset().left;
23.         var dropPositionY = event.pageY - $(this).offset().top;
24.         // Get mouse offset relative to dragged item:
25.         var dragItemOffsetX = event.offsetX;
26.         var dragItemOffsetY = event.offsetY;
27.         // Get position of dragged item relative to drop target:
28.         var dragItemPositionX = dropPositionX - dragItemOffsetX;
29.         var dragItemPositionY = dropPositionY - dragItemOffsetY;
30.         var posX = parseInt(dragItemPositionX);
31.         var posY = parseInt(dragItemPositionY);
32.
33.         alert('DROPPED IT AT ' + posX + ', ' + posY + CompName);
```

The canvas will accept items belonging to the #typePile (components) and #typePile2 (types). It means that if one of them is dropped into the canvas, the drop function will be activated.

The type name of the dragged item will be stored in a new variable and also the component name (which is only used for separating types and components).

Then the positions are taken and converted into integers.

If the dragged object is a type (CompName == null) the function „makeidcomp“ will create a random name for the new component, as the name is not really important.

```
1.  # pieditor.html
2.  if (CompName == null) {
3.
4.              function makeidcomp() {
5.                  var text = "";
6.                  var possible = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
                    xyz0123456789";
7.
8.                  for (var i = 0; i < 8; i++)
9.                      text += possible.charAt(Math.floor(Math.random() * posible
                        .length));
10.
11.                 return text;
12.                     }
13.             var compName = makeidcomp();
14.             $.ajax({
15.
16.                     url: '{% url 'create_component' %}',
17.                     type: "POST",
18.
```

```
                    data : { posX : posX,
                        posY : posY,
                        TypeName : TypeName,
                        CompName: compName},
                    success: function (component_data) {

                        window.location.href = "{% url 'select_project'
                        project=current_project %}";
                         }

            });

        }
```

Then we call a Python function for creating this component through Ajax [13], the technique that allows to make possible that webpages actualize themselves without having to download all the page again, automatically.

But after that it was needed to call another function to refresh the page (in this case for loading the project again, with the updated data) as the helper disappears just after dropping it and saving the data. Thus, after refreshing this item will be shown as a saved component instead of a type.

```
# views.py
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def createComponent(request):
    posX = request.POST["posX"]
    posY = request.POST["posY"]
    typeName = request.POST["TypeName"]
    compName = request.POST["CompName"]
    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
    myDbConnector.connect()
    myDbConnector.createComponent(posX, posY, folder_now, project_now,
    typeName, compName)
    component_data = myDbConnector.showComponents(folder_now, project_now)

    return render(request, 'shgh/pim2backup.html',
    {'component_data': component_data })
```

This function gets the position where the item was dragged, the name of the item and the name that we just set for this new component:

```
# dbpointer.py
def createComponent(self, posX, posY, foldernow, projectnow, typename, compName):

    self.dbTable = "smardism_pi_db_dev_pj.components"

    folder_name = foldernow

    project_name = projectnow

    type_name = typename
    print (type_name)

    component_name = compName
    pos_X = int(posX)
    pos_Y = int(posY)
    box = ((pos_X,pos_Y), (100,100))

```

```
    self.cur.execute(
        "INSERT INTO {0}(folder_name, project_name, type_name, component_name,
        component_date, box, pos_x, pos_y, height, width) "
        "VALUES ('{1}', '{2}', '{3}', '{4}', transaction_timestamp()
        AT TIME ZONE'{5}',  '{6}', '{7}', '{8}', '{9}', '{10}');".format(
        self.dbTable, folder_name, project_name, type_name, component_name,
        self.dataTimeZone, box, pos_X, pos_Y, 100, 100))
    self.conn.commit()
    print("Component {0} created.".format(component_name))
```

For the case when **the dragged item is a component** we call another function, „save_component“ that updates the post:

```
# pieditor.html
else {
            $.ajax({

                    url: '{% url 'save_component' %}',
                    type: "POST",

                data : { posX : posX,
                         posY : posY,
                         TypeName : TypeName,
                         CompName: CompName}
                    })
       }
```

This function „save_component“ does something similar to „create_component“ but in this case we are updating the component, not creating it.

```
# views.py
@csrf_exempt
def saveComponent(request):

    posX = request.POST["posX"]
    posY = request.POST["posY"]
    compName = request.POST["CompName"]

    myDbConnector = PiDBConnector(dbHostAddr="141.22.122.14", dbName="SHGH",
    dbUser=name_conn, dbPw=password_conn, dbdataTimeZone="CET")
    myDbConnector.connect()

    pos_x_old = myDbConnector.showPosX(compName)
    pos_y_old = myDbConnector.showPosY(compName)
    myDbConnector.changeComponent(posX, pos_x_old, posY, pos_y_old, compName,
    folder_now, project_now)

    myDbConnector.testing()
    return render(request, 'shgh/select.html')
```

It requests the current variables of the component using different functions and finally it updates the row using „changeComponent“:

```
def changeComponent(self, posX, pos_x_old, posY, pos_y_old, name, folder, project
):
    pos_X = int(posX)
    pos_Y = int(posY)
    box = ((pos_X, pos_Y), (100, 100))
    self.dbTable = "smardism_pi_db_dev_pj.components"

    self.cur.execute(
```

```
8.          "UPDATE {0}  SET pos_x='{1}' WHERE pos_x='{2}'
            AND component_name='{3}';  UPDATE {0}  SET pos_y='{4}' WHERE pos_y='{5}'
            AND component_name='{3}';"
9.          .format(self.dbTable, pos_X, pos_x_old, name, pos_Y, pos_y_old))
```

The rest of the template is written in html in order to show everything that we have set in the CSS and Javascript parts. For example it is needed to define the canvas and the two lists (components and types).

```
1.  # pieditor.html
2.  <td style="width:20%">
3.       <div id="typePile2" ></div>
4.
5.  </td>
6.  <td style="width:80%">
7.       <div id="droppable">
8.           <canvas id="droppable" width="200" height="100"
             style="border:1px solid #000000;" class="ui-widget-header">
9.
10.          </canvas>
11.          <div id="typePile">
12.
13.          </div>
14.      </div>
15. </p>
16. </td>
```

6 Improvements

Due to the lack of time and the fact that the project of the Smart Heat-Grid Hamburg is still starting, there are some things to do and to improve concerning this web server.
Here I write what I think are the main topics for continuing with my work:

Improving the PI Editor

The PI Editor is still in a starting phase. The next steps for it will be:

- Showing the correct images of the types and making possible that they have different dimensions instead of the 100x100 that they all have for the moment.
- Creating a deleting function in order to delete a dropped component.
- Create another table called „connections“ that makes a link between two or more components.

Completing the library

- Create functions for the tables „connections“ and „parameters“ (to manage some energy parameters of different projects).

Connect Web-Interface with SHGH simulation tool

- Since the development within SHGH comes closer to the execution of first simulations a connection between the web-interface and the simulation part has to be developed and implemented.

Rethink of the utilization of Django for SHGH needs

- During this thesis it was recognized that django comes with a very unique and narrow way for web interface development. This is difficult for getting started as well as complicated for the implementation of some special needs for the SHGH project, e.g. the login with database users. A switch to a more flexible tool like flask, which bring less templates but also less restrictions, can be useful.

7 Conclusion

This thesis starts an important part of the Smart Heat-Grid Hamburg project as this web application will be useful for the simulations that the researchers are going to do.

But the work has not been easy due to the lack of time and the difficulty of learning about many different tools and languages which I was not familiar with and use them for the specific tasks that the research centre required.

Thus, I could not finish some of the things that this web application needs but I could set the foundations of the web server and now it will be easier for the next researcher working on it, as it is more clear how to develope it with Django.

Also I could improve a lot my skills about programming and now I can work with more types of language, which is very important for an engineer.

References

[1] “Python,” 21 07 2017. [Online]. Available: https://www.python.org/.

[2] “Django,” 08 07 2017. [Online]. Available: https://www.djangoproject.com/.

[3] “Nginx,” 16 07 2017. [Online]. Available: http://nginx.org/en/.

[4] “PuTTy,” 21 07 2017. [Online]. Available: http://www.putty.org/.

[5] “SSH definition,” 23 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Secure_Shell.

[6] “X11 definition,” 07 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/X_Window_System.

[7] “Xming definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Xming.

[8] “PyCharm,” 21 07 2017. [Online]. Available: https://www.jetbrains.com/pycharm/.

[9] “HTML5,” 21 07 2017. [Online]. Available: https://www.w3.org/TR/html5/.

[10] “Canvas (HTML) definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Canvas_element.

[11] “Javascript definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript.

[12] “jQuery,” 21 07 2017. [Online]. Available: https://jquery.com/.

[13] “Ajax (jQuery) definition,” 21 07 2017. [Online]. Available: http://api.jquery.com/jquery.ajax/.

[14] “CSS,” 21 07 2017. [Online]. Available: http://www.w3.org/Style/CSS/.

[15] “XML definition,” 21 07 2017. [Online]. Available: https://www.w3.org/XML/.

[16] “XHTML,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/XHTML.

[17] “PostgreSQL,” 21 07 2017. [Online]. Available: https://www.postgresql.org/.

[18] “pgAdmin,” 21 07 2017. [Online]. Available: https://www.pgadmin.org/.

[19] “Primary Key definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Unique_key.

[20] “Foreign Key definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Foreign_key.

[21] “WSGI definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface.

[22] “uWSGI,” 21 07 2017. [Online]. Available: https://uwsgi-docs.readthedocs.io/en/latest/.

[23] “Virtual environment (Python),” 21 07 2017. [Online]. Available: http://python-guide-pt-br.readthedocs.io/en/latest/dev/virtualenvs/.

[24] “Ubuntu definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Ubuntu_(operating_system).

[25] «"sudo" definition,» 21 07 2017. [En línea]. Available: https://en.wikipedia.org/wiki/Sudo.

[26] “Symlink definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Symbolic_link.

[27] “Pointer definition,” 23 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/Pointer_(computer_programming).

[28] “Draggable,” 21 07 2017. [Online]. Available: https://jqueryui.com/draggable/.

[29] “Droppable object,” 21 07 2017. [Online]. Available: https://jqueryui.com/droppable/.

[30] “MySQL definition,” 21 07 2017. [Online]. Available: https://en.wikipedia.org/wiki/MySQL.

[31] “Nginx architecture,” 21 07 2017. [Online]. Available: http://www.aosabook.org/images/nginx/architecture.png.

Appendix

This Bachelor Thesis contains an appendix of the web application on a CD (disk or supplementary booklet). This Appendix is deposited with Prof. Dr. Eng. Franz Schubert.

Declaration

I declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineering and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself/ourselves independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources

Hamburg, 20. March 2017

Signature _____